Phone Based Tactile Musical Instrument: Theremin MIDI Controller

Yiannis Michael

Supervisor: Dr Gordon Dobie
Second Assessor: Dr Rory Hampson

31st March 2023

Executive Summary

The project encompassed the creation of a phone-based tactile musical instrument. Following a detailed market analysis and the initial research stage, a proposal was formed and the project objectives were defined. A Theremin Musical Instrument Digital Interface (MIDI) Controller was described as the project solution, which would have a theremin hardware interface and be able to send MIDI commands to ensure cross-compatibility with other MIDI software. A companion application would also be created to provide interaction with the hardware as well as using two of a total of three SingleTact sensors for effects on the sound output. The final product should be able to play a simple melody like "Amazing Grace".

The ultrasonic sensor chosen was the HC-SR04 transmitter and receiver, and along with the three provided SingleTacts, they were the four sensors used in the project. The sensors sent readings to the microcontroller of choice, which was the Arduino Micro and was coded in the Arduino IDE. It was chosen due to its capabilities of Universal Serial Bus (USB) communication, Analogue to Digital Conversion (ADC) and Inter-Integrated Circuit (I2C) communication. ADC was used for the HC-SR04, and I2C was used for the SingleTact sensors. All of the components fit inside the custom-designed 3D casing which had four iterations. The USB protocol was used for the hardware to communicate with the companion phone application.

The mobile app was made for the Android ecosystem and was therefore designed using Android Studio and the Java coding language. Examples provided by the Android developers were modified to fit project needs, and a custom GUI was also designed for user interaction with the app. MIDI compatibility was accomplished by using and modifying these examples. In the case of the microcontroller code, the USB-MIDI Arduino library was used to send MIDI messages to the application. The application gives four effects which can be set to the two SingleTact sensors, with the third one being mapped for the volume of the note outputs. The ultrasonic sensor provides note pitch data which is used to send the corresponding MIDI command for the app to output the specific note. By using the software and hardware together, as well as all four sensors, it is possible to play music.

The project was a success with all outlined objectives being fulfilled, as well as the completion of additional objectives which increase the applications of the final product in the real-world market of phone-based musical instruments.

Table of Contents

Table of Tables

Table of Figures

1.0 Chapter I – Research

1.1 Introduction

In this report it is explained how a phone-based tactile musical instrument was created, with detailed analysis of each step in its production. In Chapter I, the initial research stage is expanded upon. This stage gave some ideas as to which paths this project could have taken and therefore heavily informed the future decisions after this point. Chapter II documents the hardware chosen and how they are setup for the final product, as well as how they interact. Chapter III focuses on the project's software design with both application code and microcontroller code. The fourth chapter details how testing was made at each stage and how solutions were created to either completely remove some problems or to minimise their effect. Final results in this chapter are also described. Chapter V deals with discussion, conclusion, and further work.

1.2 Background

In order to fully understand and appreciate the goals of this product some general background information about musical instruments, terminology, and history of music must be explained. A term used in abundance in this report is the **Musical Instrument Digital Interface (MIDI)** communications protocol. The creation of MIDI allowed for a standard instrument language to be used during development. It was developed in the 1980s and initially proposed by the Roland founder - Ikutaro Kakehashi - to other digital musical instrument manufacturers who soon got on board and further developed the standard. Finally, it was released in 1982 and changed the way electronic instruments were developed for the next decades to come [1].

In modern music a **Digital Audio Workstation (DAW)**, is a must have tool which greatly enhances the music creation experience. DAWs are music production software that enable users to record, edit and manipulate audio files in innumerable ways. They also have the capabilities of manipulating incoming MIDI data, as well as providing some facilities to test MIDI like virtual musical instruments or custom routing of incoming MIDI commands [2].

However, MIDI and DAWs are only but recent developments in the musical sphere. The main instrument types that have existed long before them are string, wind, percussion, brass, and electronic instruments. Examples of these types of musical instruments could be guitar, flute, piano, trumpet, and synthesisers. In the case of electronic instruments, their manufacturing process differs from a traditional musical instrument. Instead of considering acoustic properties, electrical components are used to either program or create a sound output, either through the instrument itself or through another machine running a software package. The way the player interacts with electronic musical instruments is usually very similar to that of a traditional instrument, a good example of this is an electric piano or an electric guitar [3].

An electronic musical instrument very relevant to this project is the Theremin. This musical instrument was invented in the Soviet Union by Leon Theremin in 1920. In the analogue version of the Theremin, the pitch and volume are controlled by the vertical and horizontal antennae respectfully. The user can move their hands towards or away from these antennae, each of which have an electromagnetic field surrounding them, and create interference which in turn affects the outputted sound [4]. Multiple modernised iterations of the analogue theremin exist today, but other engineers have created much simpler and cheaper versions using either the same concepts or with other kinds of sensor equipment [5].

Recent developments have allowed the electronic instrument to only use MIDI and MIDI-compatible software to generate music. This eliminates the use of large and complex synthesiser circuits and allows for smaller, lighter but still musical instruments called **MIDI Controllers** which as the name suggests, are simply pieces of hardware that send or receive MIDI data to the corresponding MIDI-compatible destination. A MIDI Controller can be a device of any size and shape and wireless or wired. The only requirement is the ability to send MIDI data through whatever communications medium it is designed for [6].

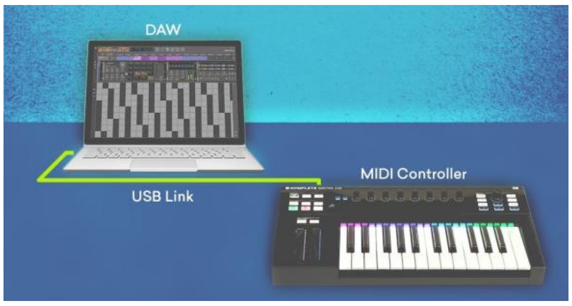


Figure 1 Example of MIDI Controller and DAW usage [1]

Some other general musical terminology used throughout this report are “arpeggio” which is defined as “the notes of the chords are to be played quickly one after another instead of simultaneously” and “octave” which is used to describe the “interval between one musical pitch and another with half or double its frequency” [7]. Some other more musical production terms used are MPC pads which stands for the Music Production Centre pads, used in a variety of modern electronic musical instruments [8].

1.3 Market Analysis

Market analysis was conducted by looking at all major music instrument retailers worldwide, and seeing what kind of products are being sold that are similar to the project description. The seven retailers used were Thomann which is based in Germany, Sweetwater and Guitar Centre which are based in the USA, Swee Lee which have a variety of locations in Asian countries, Andertons and Guitarguitar which are based in the UK, and finally Ishibashi Music which is based in Japan. Search terms such as "for iPhone", "for phone", "mobile", "portable" etc. were used to specifically showcase the products that are phone-based, or can operate with the phone as well as have other features. The target audience for this market analysis were musicians of every level whether they would be beginner, intermediate, advanced or hobbyists, as well as music producers that may not perform with these instruments but use them in the music making process.

1.3.1 Instruments

There were numerous phone compatible musical instruments that were noteworthy as shown in Table 1 . The first one was the **IK Multimedia iRig Keys 2 (and size variants)** [9]**.** Using a provided software called SampleTank SE 4, the MIDI messages sent from this MIDI controller can be outputted as sounds depending on the instrument selected, as well as other functionality customisations. Another MIDI keyboard that was examined, was the **IK Multimedia iRig Keys I/O (and size variants)** [10] which also used the "made for iPhone/iPad" tagline to signify that it was made with those users and that ecosystem in mind . A similar keyboard found on the market was the **CME Xkey Air (and size variants)** [11]**.** One thing that differentiated this from other keyboards during the market is the “Polyphonic Aftertouch” capabilities, which means that the note changes timbre while being played.

A more unique take on the MIDI controller is the **LinnStrument MIDI Performance Controller** [12] which is an expressive MIDI controller for musical performance. Finally, two very similar keyboards that were examined were the **Akai Professional MPK Mini MK III** [13] and the **Novation Launchkey Mini MK3** [14]. These keyboards encompass the idea of a typical keyboard MIDI controller that is able to be used with a phone, whilst also being portable and practical. They have the common features all of the previous MIDI controllers had and if hundreds of generic MIDI controllers were examined, they would be extremely similar to these two keyboards.

Table 1 Instruments found relating to the market analysis. Features highlight some unique aspects of each instrument.

Product Name	Features	Price	Short Description
IK Multimedia iRig Keys 2 (and size variants) [9]	• Note sustain button • Assignable knobs • Pitch bend wheel • Modulation wheel • Octave up/down buttons • MIDI IN/OUT • SampleTank SE 4	£113/£130/£149	Compatible with Android and iOS devices. Used as a MIDI keyboard to control other sounds and plugins.
IK Multimedia iRig Keys I/O (and size variants) [10]	• "Made for iPhone/iPad" • Same features as **IK Multimedia iRig Keys 2** • Built in audio interface	£201/£289	Used as a MIDI keyboard but a built-in audio interface allows another instrument to be connected and used simultaneously.
CME Xkey Air (and size variants) [11]	• "Made for iPhone/iPad", Mac/PC, Android • Octave Up/Down • Sustain Button • Polyphonic aftertouch (only one on the market) • Xkeys Plus (for iOS only)	£159	Used as a MIDI keyboard to control other sounds and plugins. Xkeys plus used as a companion application for customization.

Table 2 Continuation of previous table.

Product Name	Features	Price	Short Description
LinnStrument MIDI Performance Controller [12]	• Reads and sends MIDI data on pressure, left-right, front-back, release, velocity • Step-sequencer • Customisable Note pads • All previously mentioned features (eg. sustain and octrave buttons) included.	£1399	A more unique take on the MIDI controller. Used as a MIDI keyboard to control other sounds and plugins; highly customisable.
Akai Professional MPK Mini MK III [13]	• Standard MIDI controller • Arpeggiator • 4-way joystick for dynamic pitch/modulation control • 8 MPC pads • 8 assignable knobs	£77	Used as a MIDI keyboard to control other sounds and plugins. A generic MIDI controller with most of the common features.
Novation Launchkey Mini MK3 [14]	• Standard MIDI controller • 16 performance tabs • Arpeggiator • Pitch/Modulation strips	£60	Used as a MIDI keyboard to control other sounds and plugins. A generic MIDI controller with most of the common features.

Figure 2 Image of the Novation Launchkey Mini MK3 [14]

To summarise, on the market there is a limited number of phone-based musical instruments, and those that are phone=based, are MIDI controllers. There a few that would be suitable to use with a phone or in generic portable applications. Those discussed above, all share remarkably similar characteristics with only a few differences that distinguish them. In the case of the Linnstrument, it is seen that there is demand for other unorthodox MIDI controllers that can perhaps add unique options to the performance, or the creative process.

1.3.2 Audio Interfaces

One of the most popular peripherals musicians have for the phone is the audio interface [15]. This is not a musical instrument itself, but it is essential for the musical creation process or the musician's performance. It sits between the musician's audio signal and the phone so as to allow for a data conversion to take place and for recording to happen. The recording can either be real time to an audience or saved to be processed at a later date.

Two very popular audio interfaces are the **IK Multimedia iRig Pro I/O (£149)** [16] and the **IK Multimedia iRig Stream (£88)** [17]. These devices provide the musician with a minimal audio recording setup in a variety of scenarios that require portability and short start-up time.

The market research carried out revealed the popularity of these devices, whilst also making evident that the modern musician is looking to incorporate their phone into their audio setup. Doing so allows for greater portability and less heavy gear to carry from venue to venue.

Figure 3 Image of the IK Multimedia iRig Pro I/O [16]

1.3.3 Apps & Software

There are nearly hundreds of apps available for both Android and iOS ecosystems catered for musicians, as shown in Table 3. Most of these apps revealed to be MIDI compatible apps where they either send or receive MIDI messages. This is where a MIDI controller comes into play, meaning that the apps can process the data sent by the MIDI controller in whichever way they are set to do so.

Table 3 Applications found during the market analysis.

App Name	**Description**	**Price**
Wireless Mixer – MIDI [18]	Can send or receive MIDI messages over WiFi and allow the user to blend their track with other loaded tracks.	Free
TouchDAW [19]	Wireless DAW/MIDI controller that can connect to other MIDI compatible devices wirelessly and either send or receive MIDI data.	Free
MIDI Commander [20]	Extremely customisable MIDI controller in an app format.	Free
MIDI Keyboard [21]	Simple synthesiser/keyboard app that allow the user to play music or to send/receive MIDI data from other sources.	Free
Fluid Synth [22]	Provides the sound libraries required to create music and a MIDI controller is used to play them.	Free
FL Studio Mobile [23]	Advanced DAW experience on mobile.	£14
Cubasis 3 [24]	Advanced DAW experience on mobile.	£27
Sample Tank 4 [25]	a similar concept to Fluid Synth where it acts more as a sound library rather than a DAW.	Free

To conclude, the mobile app market for musicians is dominated by MIDI compatible apps. This shows how MIDI has been adopted and used in the handheld world, which is far from its initial beginning in 1982 in wired electronic musical instruments. Apps use MIDI in a variety of ways, from sending or receiving MIDI commands, to processing these commands and outputting the sound of a real musical instrument. They complement MIDI controllers and can be thought of as companion apps that expand the musicians' possibilities when using MIDI with a mobile device.

1.3.4 Hardware & Software Packages

Market analysis reveals a variety of devices which are accompanied by an app that allow the hardware to flourish and show its full potential. Such packages like the **LUMI Keys** [26] and the **Joué Play** [27] have hardware and software that complement each other, but come at a cost of not working well as general purpose MIDI controllers. LUMI Keys and the accompanying app are used predominantly as a learning tool while the Joué Play is marketed more as a music creation tool. Even though the Joué Play is MIDI compatible, packages like it usually do not have the capabilities to be used in a professional environment and are created to help hobbyists or beginners understand the music production environment.

1.3.5 Theremin

To provide further background and to further analyse the market, a look at the Theremin market is necessary. Theremins have existed for a long time with music being written specifically for them, whether in movie scores [28] or rock songs [29].

The **Doepfer A-178 Theremin** [30] is a modular synthesiser module that shows how small a theremin can be with dimensions of 40.3 mm by 40 mm (W x D). A more typical theremin is the **Moog Theremini** [31] with its two antennas and extra features, such as variable quantisation of notes as well as MIDI USB connection. Another product would be the **Moog Clavarox Centennial Theremin** [32]**,** which is a premium theremin that also has a MIDI USB connection.

In general, theremins exist in a variety of sizes with some customisation features and connections. Using MIDI with a Theremin is an existing concept but it does not exist exclusively in a MIDI controller format.

1.3.6 Market Analysis Conclusions

To summarise, the market analysis has shown very clear patterns in terms of what type of musical instruments are used together with a phone. MIDI is a common feature in all musical instruments examined, and therefore MIDI controllers dominate the market. Most controllers of this nature are in the piano keyboard format, due to it being one of the most intuitive and easiest instruments to quickly learn. Other forms of the MIDI controller exist but they usually do not stray far from what is expected, even though there is a market for that as well.

The demand for audio interfaces showcases how musicians are very keen in using their mobile devices for their music setups, whether that would be production or performance. The largest section of this market is catered towards the iOS ecosystem, due to its standardisation and quality assurance, with Android still having a share. A variety of apps exist on both marketplaces that are made for musicians. Apps could encompass playing music or sending/receiving MIDI messages. Companion apps also allow for custom hardware to shine and showcase its capabilities; even though they are not used in a professional context, they still show potential.

1.4 Proposal, Aim and Objectives

The conclusions drawn from the background research and market analysis have lead toward the proposal of a **Theremin MIDI Controller (TMC)** with a companion Android application. The connection between the two devices will be using the Universal Serial Bus (USB) standard as well as the Java language for app development. It will have a minimal 3D design with simple app customisation. The hardware will be MIDI compatible which will allow for use with all MIDI compatible software on the market. Discussions of each hardware or software decision will be made in their corresponding chapters.

As stated in the Statement of Intent and Interim Report, the project objectives are:

- Theremin hardware interface using an ultrasonic transmitter and receiver.
- Make the instrument able to send MIDI commands in order for it to be used in other applications.
- Application created to provide phone interaction with instrument.
- Ability to play a simple song like “Amazing Grace”.
- Implement three SingleTact sensors, with two designated for effects.

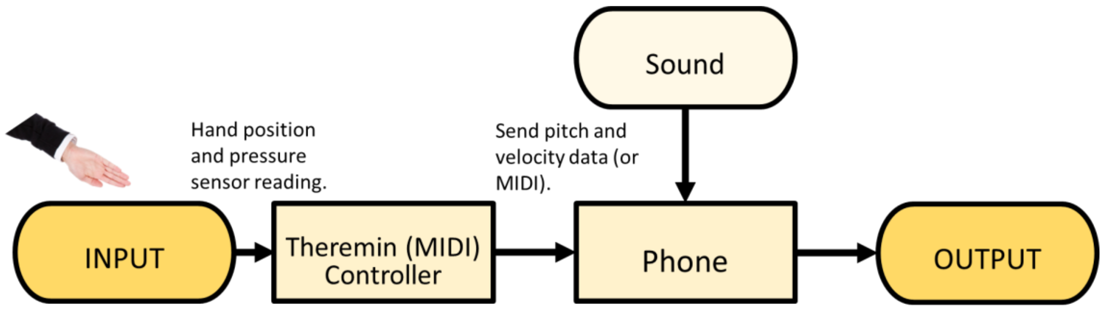

Sound
Hand position and pressure sensor reading.
Send pitch and velocity data (or MIDI).
INPUT
Theremin (MIDI) Controller
Phone
OUTPUT

Figure 4 High level diagram of the Theremin MIDI Controller

The theremin functionality will be accomplished with a distance sensor and the SingleTact sensors which will employ the user’s hand for input. This input will be processed through a microcontroller and sent to the phone through a wired connection. The app will then process this transmitted data and give the corresponding output.

1.5 Planning

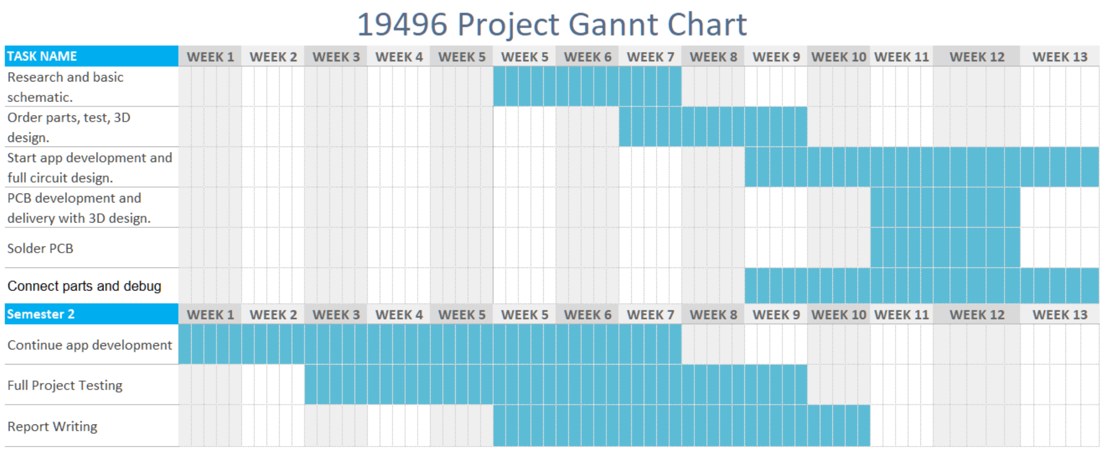


Figure 5 Initial Gannt Chart. Each box reflects a day in a week. Error in semester two week five label, but cannot be changed since it is an old image.

An initial Gannt chart was created (Figure 5) and time was allocated based on it. Three weeks were allocated for research and to create a basic schematic. Three weeks were also allocated to order parts and to start designing the casing for the product. Based on this initial chart, at Week 9 the app and full circuit design should have started and initially take around four to five weeks or until the first semester ends. The same amount of time would be allocated for connecting the parts and debugging the project as it progresses. While both previous tasks are happening, PCB development, soldering and 3D design of the casing will also be occurring. In semester two, the app development would continue until Week 7 with full project testing starting in Week 3. Both will finish in Weeks 7 and 9 respectively. Report writing should start in Week 6 (mistake in figure) and continue until the end of the project. This chart was updated roughly around Week 9 of semester one so this structure was roughly followed until then.

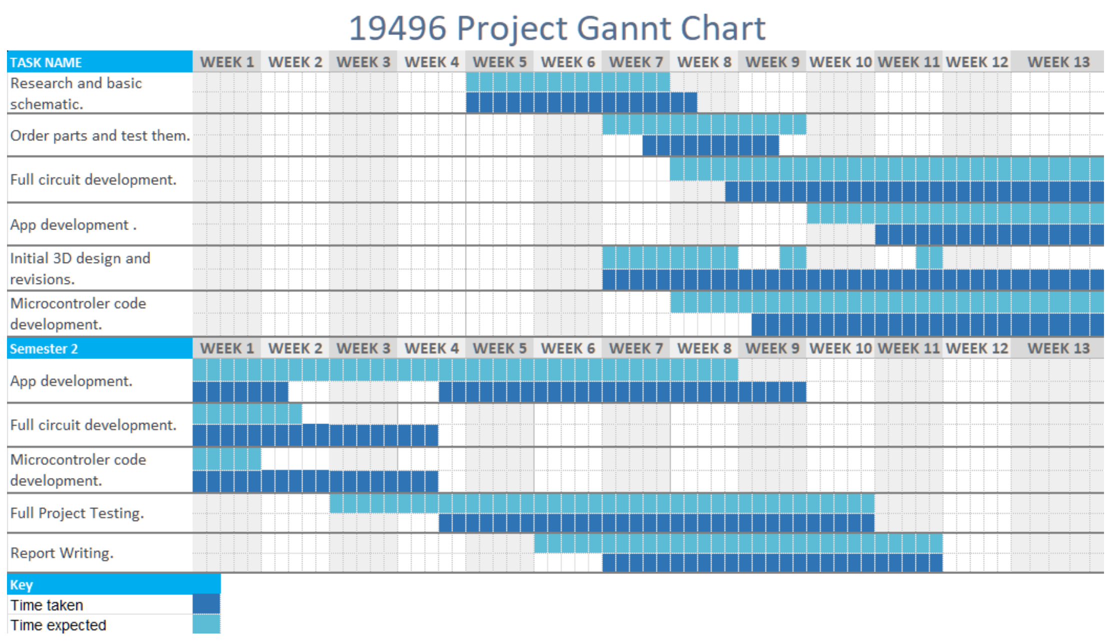


Figure 6 Final Gannt Chart. Dark blue is time taken and light blue is time expected. Each box represents a day

The Gannt chart was then revised before the end of the first semester (Figure 6), and due to having a better understanding of how the project will progress, more tasks were created and time was allocated more effectively. After the initial research stage, it was concluded that no PCB would be required, and that the project would take a more software-centric approach to make it easier to manage in the later stages were time is critical. Ordering of components or finding problems with compatibility near the end of the project with a hardware-centric approach could prove extremely difficult to resolve.

Same time was allocated for initial research and ordering parts. Then, full circuit development would start in Week 8 and continue until around semester two Week 2. One section each was allocated for the mobile application development and the microcontroller code development, with the microcontroller section taking around the same time as the full circuit development. App development should start a week after full circuit development and continue until semester two Week 8. The 3D design should start early on in semester one Week 7 and finish by the end of the semester. Full project testing should start near the beginning of semester two and around one week before the report writing in Week 10. Report writing should also be done at the same time as full project testing and app development but should start in Week 6 of the second semester.

Overall, the planning of the project allowed for extra time when setbacks occurred. Major obstacles of note were in Weeks 2 to 4 of semester two where there was an issue with MIDI detection on the Android phone, so microcontroller code development and full project testing was extended by two weeks. The app development also was having little progress during the same timespan due to issues with the Java code. Another setback was the learning curve of using 3D Computer Aided Design (CAD) software since no previous experience was had with such design applications. These issues are expanded upon in their respective sections, but in the end they did not severely affect the completion of the project objectives and everything was completed on time.

1.6 Budget

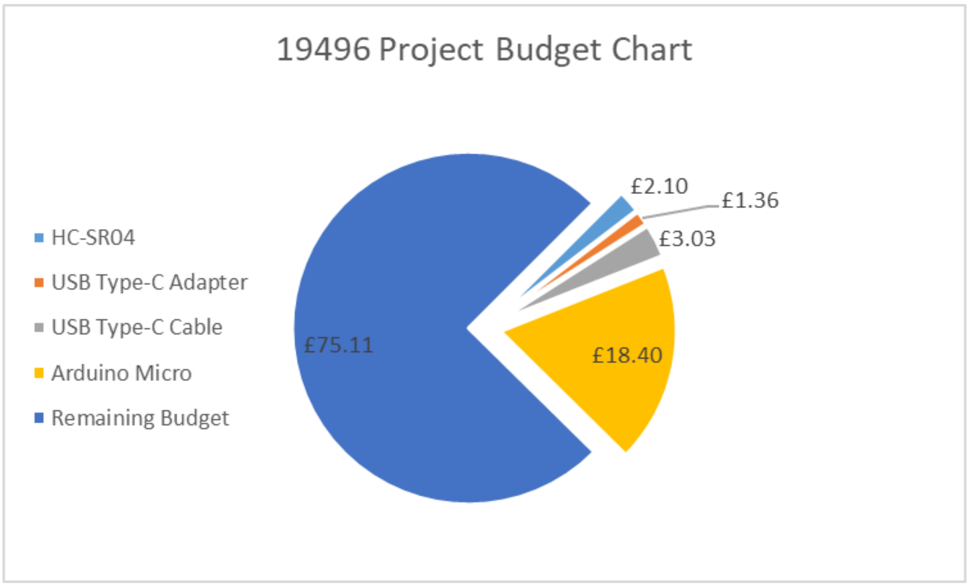


Figure 7 Chart showing budget allocation

Figure 7 showcases where the budget was allocated, as well as the prices for the components (HC-SR04 [33], USB Type-C Adapter [34], USB Type-C cablec[35] and Arduino Micro [36]) and how much they took from the budget. With this chart it is safe to conclude that this project was well within budget.

1.7 Project Motivation

This project aligns with my career aspirations which are to work in musical instrument production as well as embedded system design. It also will enhance my skills in product design and general programming. There is nothing like it on the market and therefore it is also an area worth exploring and expanding upon.

2.0 Chapter II – Hardware

2.1 Component Selection

Choosing which components to order and use, was based on which path the project would take. If more advanced hardware were used then more unusual hardware would be chosen for the project. For example, a microcontroller programmed with In-Circuit Serial Programming (e.g. MSP430F5500 [37]) could have been used, along with a USB Host integrated circuit (e.g. FT312D [38]) and custom PCBs and soldered USB connectors. This route was decided on being too time consuming due to completely new approaches to programming hardware and data communication, as well as being harder to debug in late stages of development where a wrong or faulty component could jeopardise the entire project.

The route taken and detailed in this report is the advanced software route, where simpler hardware was chosen and instead using advanced or new programming techniques to complete the project objectives. This would make the product easier to debug and test in the late stages since the hardware could be completed early on, it would also be cheaper due to simpler hardware being less expensive and less likely to have incompatibilities. Libraries and examples of difficult programming challenges also exist online and could therefore be used and further decrease the difficulty of project objective completion.

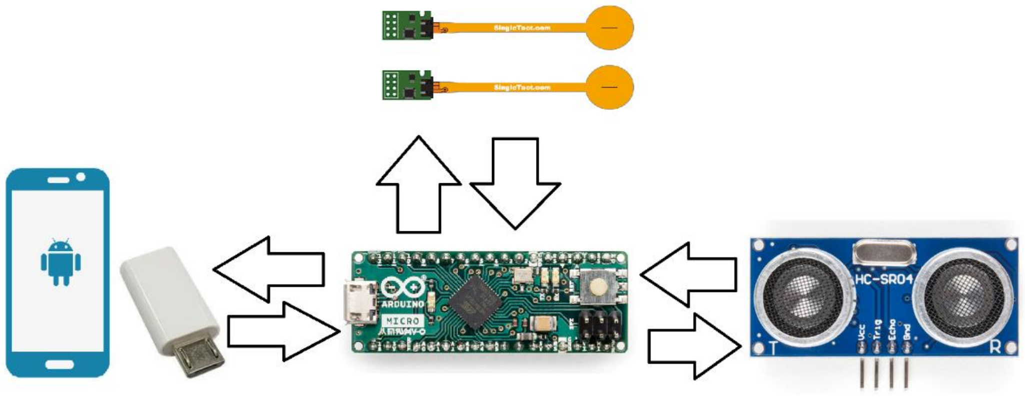


Figure 8 Diagram of TMC components and their interaction

2.1.1 Distance Sensor

The distance sensor selected was the **HC-SR04 Ultrasonic Distance Sensor**. It was chosen due to being a receiver/transmitter combination in a single PCB. The working current and voltages (30 mA and 5 V respectively) are sufficient to be powered by a small microcontroller. The range of 2 cm to 4 m is also more than enough for simple hand detection. The V_{cc} , Trigger, Echo and Ground pins have headers which can be connected to a microcontroller with male pin headers using female to female header cables. An Analogue to Digital Converter (ADC) will be used to convert the readings to a useable format in the microcontroller code [33].

2.1.2 SingleTact Sensors

These pressure sensors were a must for the project due to one of the objectives being to use at least two SingleTact pressure sensors, but an extra one was added bringing the total up to three to make the hardware a more complex. A choice of 4.5 N, 45 N and 450 N had to be made, and with using a food scale to the strength required for the specific ratings, the 4.5 N model was selected. One SingleTact sensor has a supply voltage and current of 3.7 V to 5.5 V and 2.7 mA respectively. The communication method used to receive data from the SingleTacts was Inter-Integrated Circuit (I2C). [39]

2.1.3 Microcontroller

The microcontroller chosen was an Arduino Micro. This decision was influenced by the need to have USB data transmission, an ADC for the Ultrasonic Sensor data and to use I2C for the SingleTact sensors. The Arduino Micro checks all the boxes, as well as being one of the smallest units in the Arduino range. It has seven pins for ADC conversion which one will be used to read data from the HC-SR04.

Table 4 Different requirements for the components used

Component	Amount	Current Required (mA)	Voltage Required (V)
HC-SR04	1	30	5
SingleTact Pressure Sensor	3	2.7	3.7 – 5.5

Based on Table 4, a voltage of 5 V and a total current of 38.1 mA are required to power all components. The Arduino Micro can provide 5 V and 50 mA max from its 5 V pin, making it ideal for supplying power to the components. [40]

2.1.4 Mobile Phone System

Due to having a personal device using the Android operating system, it was preferred to use an Android phone as the type of mobile phone the final product would connect to. Therefore the Android phone used was a OnePlus 9 with Android 13.

2.2 Wiring

Using female to female or female to male header cables was the main way this project was wired. All cables were custom wired to accommodate the project's needs for a compact system. A wired USB Type-C cable [35] and a USB Type-C Female to Micro USB Male Adapter [34] was used in order for the specific cable to connect to the Arduino Micro's USB port. A USB Type-C connection between the microcontroller and the phone was chosen due to the growing market of USB Type-C devices. Using an adapter only provided the reversible benefits of this type of USB and not the data transmission or power benefits, this is due to the Arduino Micro only being USB 2.0 capable [40].

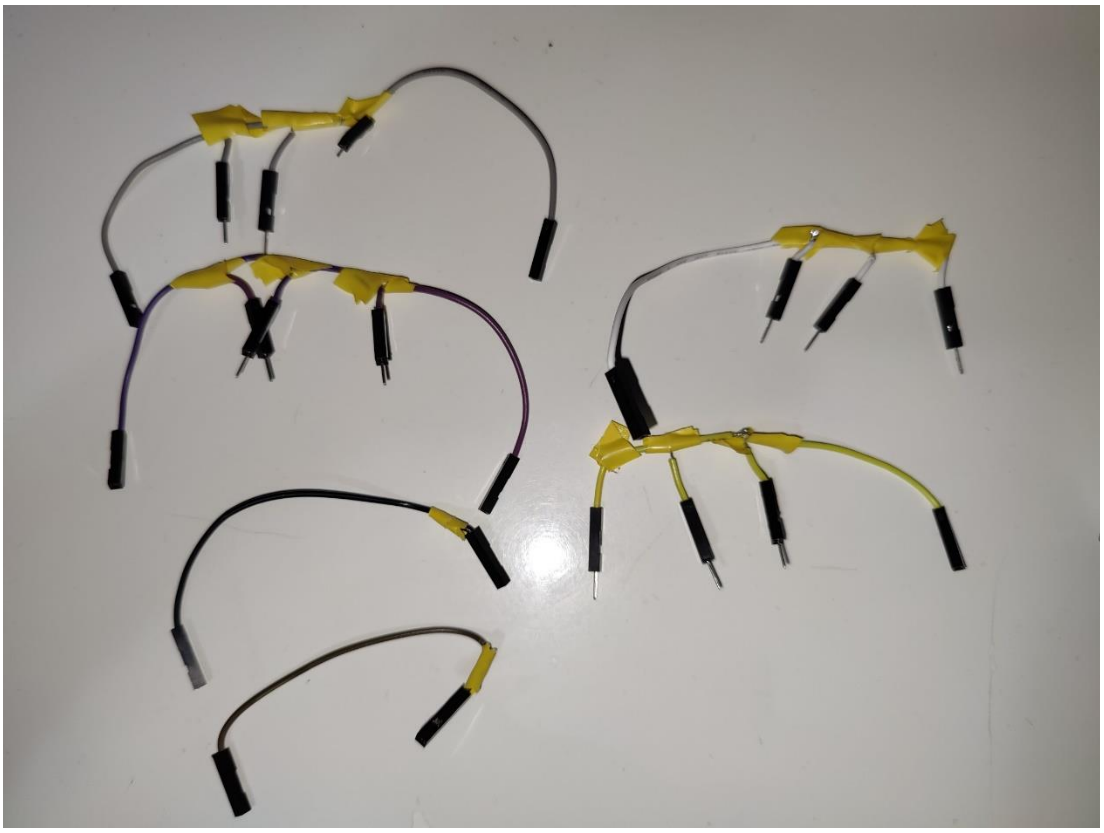


Figure 9 Custom soldered wiring. From top left to bottom left, going from left to right are the following wires: voltage, ground, SDA, SCL, Trigger and Echo.

2.2.1 Voltage and Ground

Wires for providing a voltage source and grounding were custom wired using six male header cables and four female header cables. Insulation tape was added to prevent shorting. In the voltage wire, the three male header cables are for the three SingleTact sensors, and one of the female headers is for the HC-SR04 voltage pin. The remaining female header is for the Arduino Micro 5 V pin which provides the required voltage supply for all the components to function. Similarly, for the ground wire there are the same connections except it connects to the Ground pin of the Micro [40].

For the SingleTact sensors a male header is inserted into pin 1 for the voltage and into pin 8 for the ground [39]. Whereas for the HC-SR04 the female header is used and is inserted into the V_{cc} and Ground pins, for providing a voltage and groundrespectively [35].

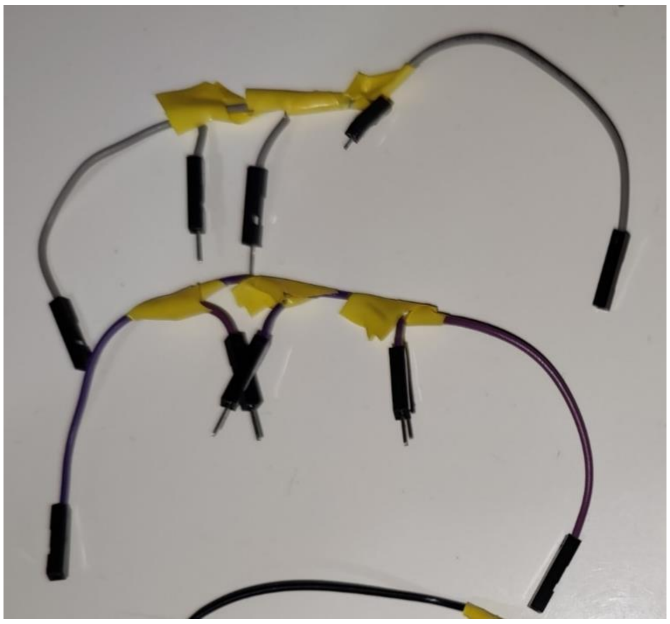


Figure 10 Image of the Voltage (bottom) and Ground (top) wires

2.2.2 Data transfer

Communication between the sensors and the microcontroller happens with four wires. Custom wires were made for the Serial Clock (SCL) and Serial Data (SDA) connections required for I2C to happen. The three male headers go into the SCL and SDA pins of the SingleTacts. Another pair of wires is used to connect the HC-SR04 Trigger and Echo pins to the corresponding pins on the Arduino Micro.

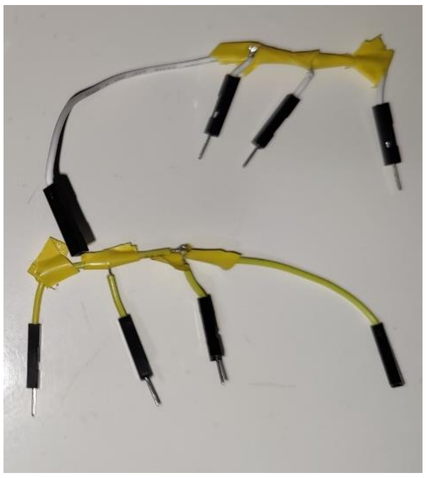


Figure 12 Image of SDA and SCL wires for the SingleTact sensors

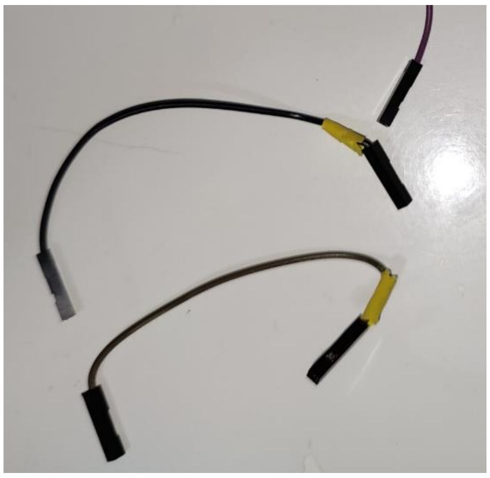


Figure 11 Image of Trig and Echo wires for HC-SR04

The male header of the SDA and SCL wires are inserted into pins 6 and 3 of the SingleTacts respectively [39]. These two wires allow for I2C communication between the microcontroller and the sensors. Pin D2 is the SDA pin for the Arduino Micro and D3 is the SCL pin. The corresponding female pin header is inserted into these two pins to allow for the connection to be complete. For the HC-SR04, the wire connected to the Trig pin gets connected to pin A1 of the Micro and the Echo pin is connected to the A0 pin. The output of the pins mentioned was configured in the Microcontroller code.

2.3 3D Design

The casing was designed using Autodesk Fusion 360 which is 3D Computer Aided Design (CAD) software. Four main prints were made with a final Print 5 being planned if the project finished early. Ready 3D designs for the Arduino Micro [41], HC-SR04 [42] and the HC-SR04 holder [43] were downloaded from the appropriate sources. Custom designs were created for the SingleTact sensors and the USB Type-C adapter by measuring them and creating diagrams (see Figure 18 and Figure 17). These components were needed in order to understand the space they would occupy and how they would all fit together. The design focused on portability and ergonomics while it was constrained by the Arduino Micro headers, space taken by the wires, and the flat surface required by the HC-SR04 and SingleTacts.

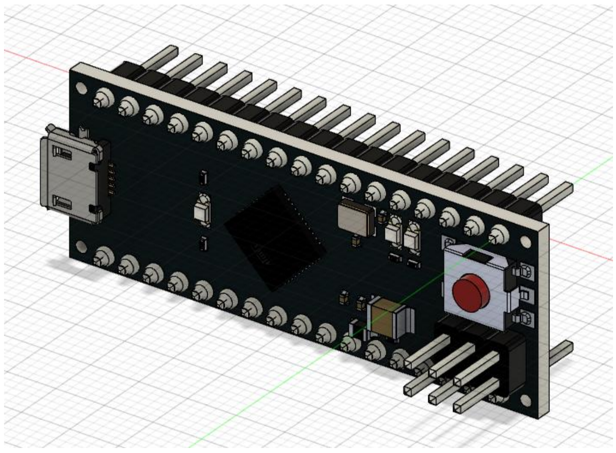


Figure 14 3D CAD model of the Arduino Micro [40]

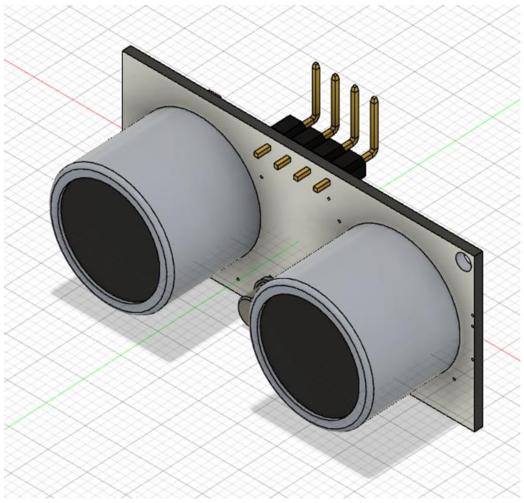


Figure 13 3D CAD model of the HC-SR04

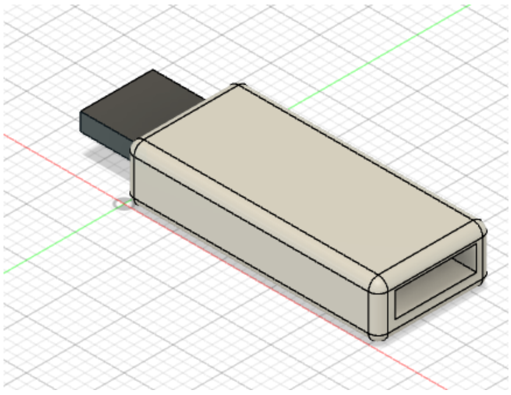


Figure 16 Custom 3D CAD model of the USB Type-C Adapter

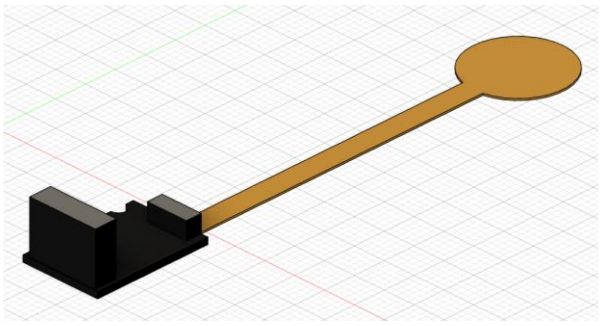


Figure 15 Custom 3D CAD model of a SingleTact Sensor

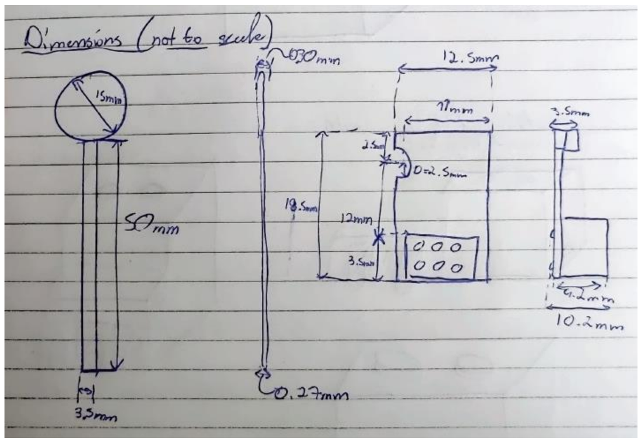


Figure 17 Initial measurements and sketches of a SingleTact Sensor

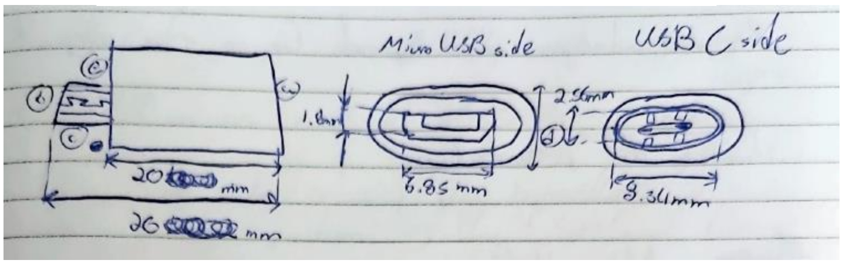


Figure 18 Initial measurements and sketches of USB Type-C Adapter

2.3.1 Initial Concepts

Shown below are some initial concepts that were first drawn, considered, and then designed in Autodesk Fusion 360 for a better understanding of the dimensions, and ergonomics of the product.

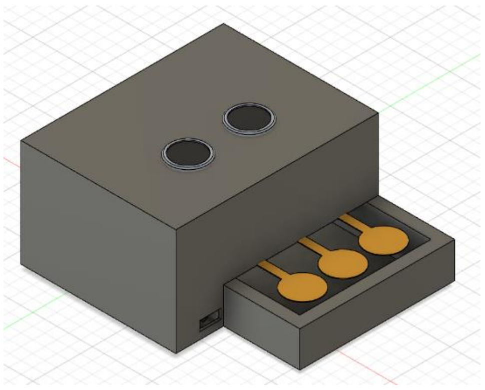


Figure 20 TMC Concept A

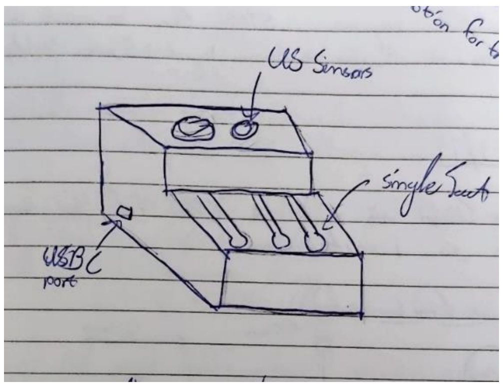


Figure 20 TMC Concept A Sketch

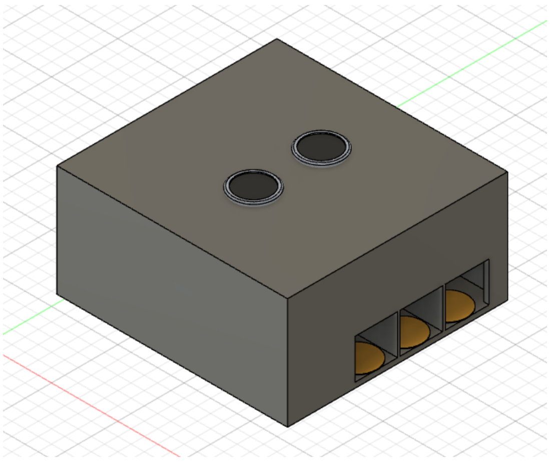


Figure 24 TMC Concept C

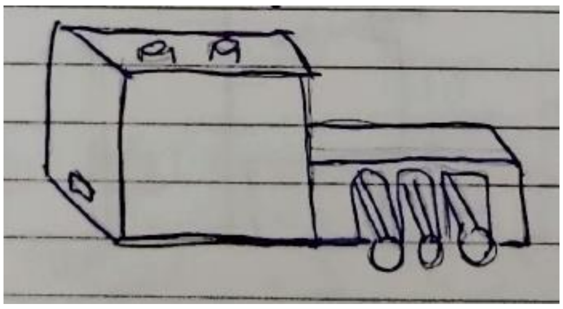


Figure 24TMC Concept B Sketch

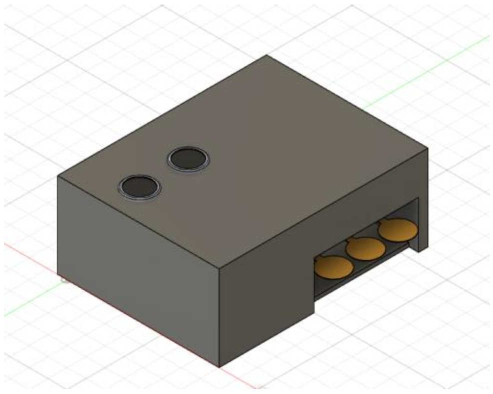


Figure 22 TMC Concept B

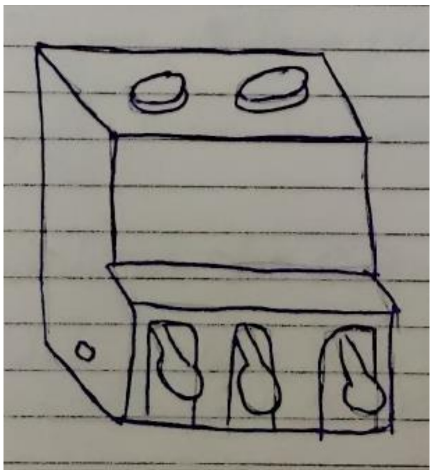


Figure 22 TMC Concept C Sketch

After careful consideration, TMC Concept C (Figure 24) was the one declared most suitable for printing. The SingleTact sensors would be on the bottom as shown, and the HC-SR04 would stick out from the top to allow to distance detection. The Arduino microcontroller would be to the left of the pressure sensors with a USB port at the back of the casing. Some modifications had to be made in order for the components to be fitted inside and allow for further inspection. Therefore, this design was adapted with a sliding-up sensor front cover and a top lid which resulted in Print 1.

2.3.1 Print 1

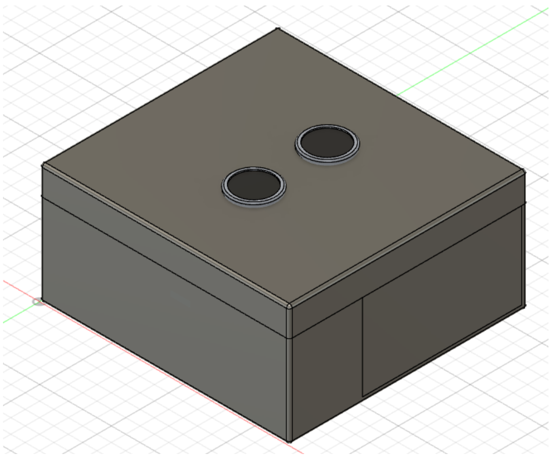


Figure 25 3D CAD Design of Print 1

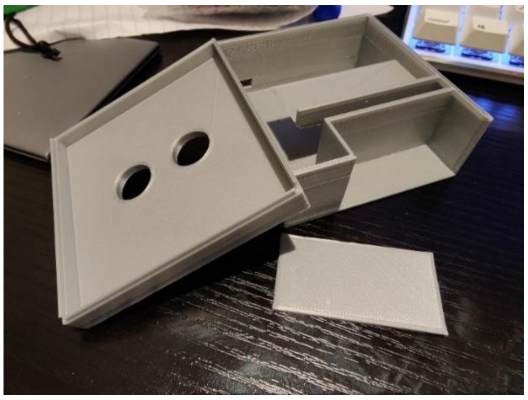


Figure 26 Print 1

Print 1 (Figure 26) was the first printed concept which gave a lot of insight into how the product should be improved in future prints. It had a lot of issues which meant improvements and a second print was to be scheduled. The issues include insufficient space for the Arduino Micro, hole for USB port was too small, HC-SR04 sensor hole was too small, no way to have SingleTact sensors stay in place and no holes at the front for them to stick out and be useable. The dimensions for Print 1 were 87 x 83 x 42 mm (W x L x H). These dimensions remained the same for Prints 2 and 3.

2.3.2 Print 2

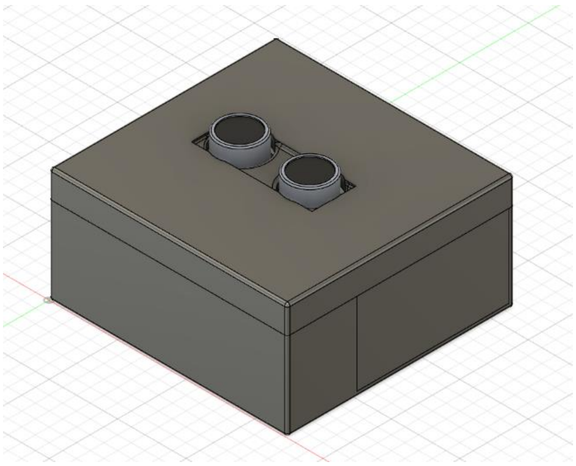


Figure 28 3D CAD Design of Print 2

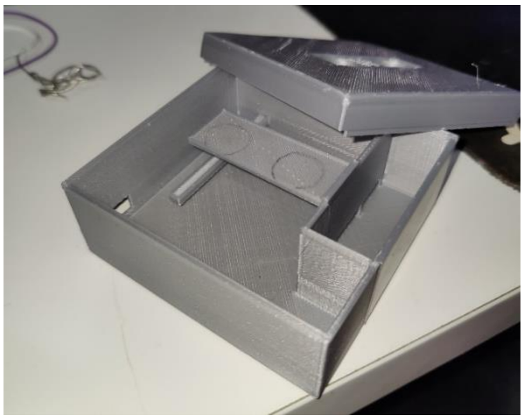


Figure 27 Print 2

For Print 2 (Figure 27), the issues of Print 1 were resolve but other issues also arose which prompted Print 3. The HC-SR04 shelf was rotated by 90◦ to make more space for the microcontroller headers and a stop bar was added to hold the SingleTact sensors in place. Bigger holes for the USB port, the HC-SR04 and SingleTact sensors were also designed.

The issues were that the Arduino Micro and the HC-SR04 not being able to remain horizontal. This was especially important for the HC-SR04 since it should be completely horizontal in order to accurately measure distances. The box shape was also not aesthetically pleasing and therefore some aesthetic changes would have to be done in the next print.

2.3.3 Print 3

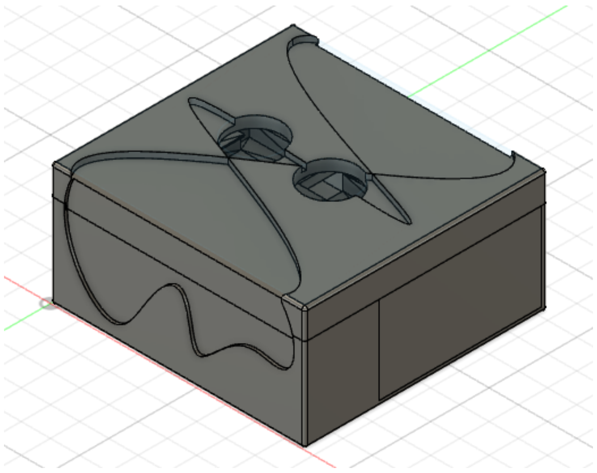


Figure 30 3D CAD Design of Print 3

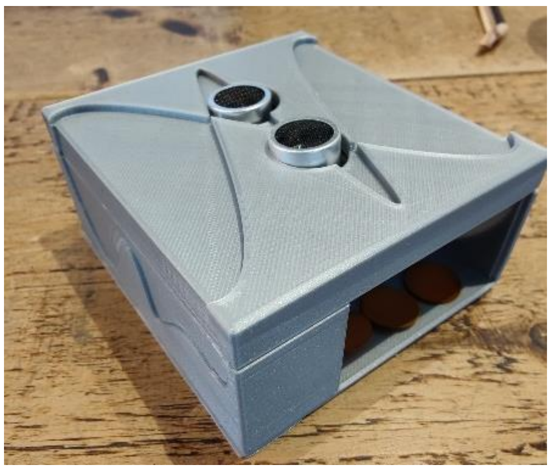


Figure 29 Print 3

Print 3 (Figure 29) improved on the issues mentioned in Print 2, however more issues needed to be addressed. The HC-SR04 holding bracket () downloaded from Thingiverse [43] allowed for the ultrasonic sensor to be stable and horizontal at all times. A small block was also added for the Arduino to sit on and be able to stand horizontally. Some aesthetic changes were also made to the outside of the casing to make it more aesthetically pleasing.

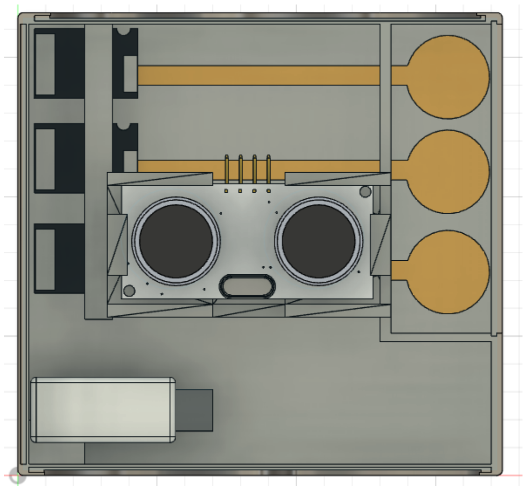


Figure 31 Top View of Print 3. The new holding bracket can be seen in the middle and the block for the Arduino can be seen at the bottom left.

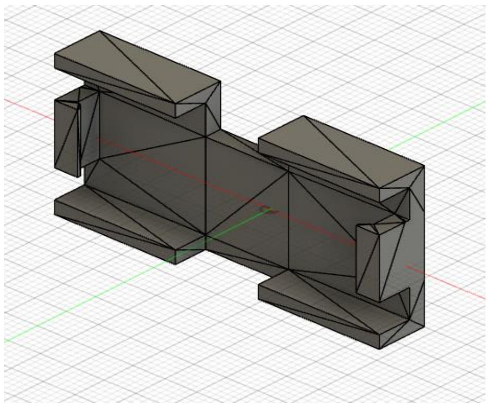


Figure 32 HC-SR04 Holding Bracket from Thingiverse [42]

The issues with this print were that the walls were now too thin due to the new outside design. This made the lights of the Arduino be able to shine through and the casing now felt fragile. Using this print, it was discovered that more space for the wiring was required, therefore it had to be made bigger and with thicker walls. The sliding sensor cover at this point could only be removed upwards, but that meant the user always had to take off the cover and access the wiring in order to play with the TMC. As this is a musical instrument and a product, this had to be remedied by creating a sliding cover that goes to the right which would allow to open and close the sensors without exposing the wires.

2.2.4 Print 4

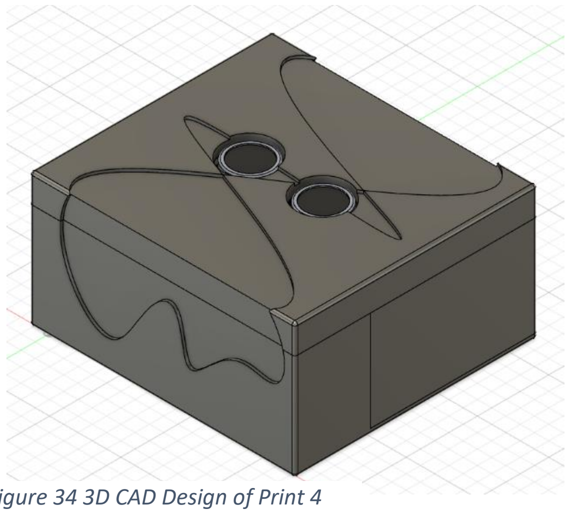


Figure 34 3D CAD Design of Print 4

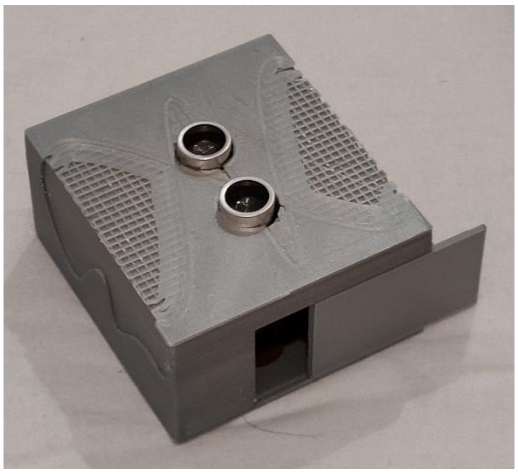


Figure 33 Print 4

Print 4 proved to be the best and final print. It was a bigger version of Print 3 with a sliding sensor cover, an obstacle to keep the microcontroller in place and thicker walls. These new features are visible in the top view when comparing Print 3 (Figure 31) with Print 4 (Figure 35). The finalisations made to this print after it was printed were double-sided tape added to keep the SingleTacts from moving, and hot gluing the Arduino in place so that no movement occurs. There are no major issues to this print but improvements could be made and some are noted in Print 5. The dimensions of Print 4 are 92 x 85 x 47 mm (W x L x H) with height being 57 mm with the HC-SR04.

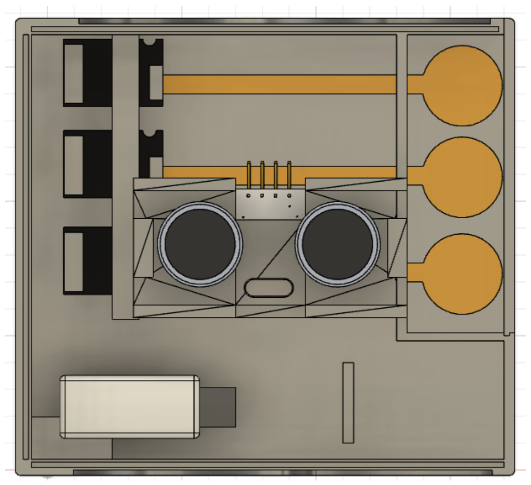


Figure 35 Top view of Print 4. Obstacle for the microcontroller can be seen at the bottom as well as the thicker walls and the new sliding sensor cover

2.2.5 Print 5

Print 5 was not completed due to other project objectives taking priority. Print 5 would have incorporated an indicator LED to show to the user if data is being read or sent. A screw would also be situated at the top lid to further prevent the user from opening the TMC and exposing the wires. Print 4 is acceptable and fully incorporates all the needs of the final product so a fifth print was not needed.

2.4 Hardware Issues

Even though the hardware helped fully accomplish all project objectives, some issues did occur and had to be dealt with accordingly. The issues with the casing of the TMC were noted in section 2.3 for each print specifically.

One critical issue was with one SingleTact sensor not outputting the expected readings. This was discovered when it was being tested during code development and more detail as to how it was discovered will be in Chapter IV. To resolve the issue, a new sensor was given by the manufacturer. Moreover, the SingleTact sensors were not calibrated and also moved during use. Double sided tape was used to keep them in place and the built-in Arduino “map()” function was used to calibrate their values as best as possible. This will be detailed in the Software development section.

Another issue of note is the accuracy of the HC-SR04 when using a surface such as a hand that is irregular and soft. This specific sensor is also very sensitive to noise from other reflections not from the object being measured. There was an attempt to combat this by using a rolling average when gathering measurements. This will be examined further in Chapter IV.

The Arduino Micro also was prone to moving around in the casing. It was remedied with a small sponge but in the end it was hot glued in place. On the wires, glue on the insulating tape worn out through time and this can be helped by using a better insulation tape. The wires for the Trigger and Echo pins also had a weak point at the solder joint which prompted a re-soldering of the wires. Both were soldered in a similar manner as before but more in the middle of the wire, and for one of them a heat shrink was used (Figure 36).

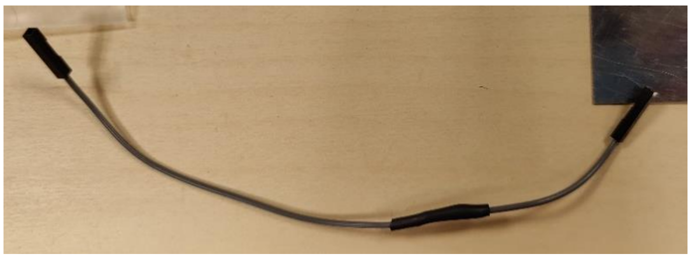


Figure 36 Re-soldered wire. Moved the solder joint and a heat shrink was used.

2.5 Component Alternatives

One other possible solution for the distance sensor, was the **Devantech SRF08 Ultrasonic Range Finder** [44]. This sensor provides with I2C communications with the microcontroller unit, but due to the massive price difference, and to help keep the I2C bus as free as possible from unnecessary transmissions, the HC-SR04 was chosen instead.

Multiple Arduinos were taken into consideration when deciding which microcontroller to use. Other suitable models were the **Arduino Uno rev3** [45] and the **Arduino Due** [46] with its extra programmable USB port, but to make the final product more portable, and to not waste extra features, the Micro was chosen instead.

3.0 Chapter III – Software

3.1 Communication Methods

Multiple communication methods were used in this project. Specifically there were two ways the sensors communicated with the microcontroller and two ways between the phone and the microcontroller.

3.1.1 USB and MIDI

These two methods were used together at the same time for communication between the mobile phone and the Arduino Micro. To accomplish such a task, the Arduino MIDI-USB Transport Library [47] was used. This library incorporates and handles the bulk of the work, for both the MIDI and USB functionality required, for the phone and the Arduino to communicate with a wired USB connection to send MIDI data.

MIDI data is usually referred to as MIDI messages or MIDI events when talking about a message at a particular point in time. They carry the information about which MIDI command is being sent, as well as one or two values related to that command. For example, when sending a Note On MIDI Message, the first parameter is the value "0x9" and the next two parameters are the note number (or note value) and the intensity of the note which is called velocity [48]. There are hundreds of MIDI commands and some are even custom made for a specific MIDI instrument. MIDI even supports up to 16 channels for multiple MIDI event handling at the same time, but this project only used channel 1 (or 0) for simplicity [1]. The event types used in this project and their parameters are shown in Table 5.

Table 5 MIDI event types used in the project

Event Type	Event Value (hexadecimal)	Parameter 1	Parameter 2	Description
Note On	0x9	Note Value	Velocity	Used to turn on or off notes based on their note value. Velocity determines the intensity of the note. Parameters 1 and 2 range is from 0 to 127 [49].
Note Off	0x8			
Pitch Bend	0xE	Value LSB	Value MSB	Used to change the pitch of the note in the current channel. Each parameter value is 7 bits long which makes up a 14-bit number. Range is from 0 to 8192 in each value. A total value of 8192 means no pitch bend [48].
Control Change	0xB	Controller Number	Controller Value	Controller numbers used are only the ones listed. Sent when it is needed to change the MIDI channel state [50].
		0x07		Volume – By default changes the volume in an application. In the TMC app it is mapped to the velocity.
		0x0C		Effect Control 1 – Is usually mapped to an effect by default in many applications. It had a similar application in the project.
		0x0D		Effect Control 2 – Same as Effect Control 1.

3.1.2 I2C and ADC

Inter-Integrated Circuit and Analogue to Digital Conversion are the other two communication methods used in this project. I2C is used for reading data from the SingleTact sensors and ADC is used when receiving readings from the HC-SR04.

The I2C serial communications protocol is used for short distances with a single or multiple devices. Two wires are required to exchange information between two devices, with multiple wires used when multiple "peripherals" or "controllers" exist on the same bus. A peripheral can communicate with a controller where the controller reads or writes to or from the peripheral. One wire required is called the Serial Data and the other is called Serial Clock. The clock signal is sent by the controller to synchronise all of the peripherals on the bus together. Addresses are assigned to the devices connected to the bus so that data reaches the correct destination. Pull-up resistors are required to keep the bus signal to high so that no unnecessary power dissipation occurs and no damage is caused to the connected devices [51].

This I2C functionality withing the Arduino Micro is handled by the **Wire** [52] library. Peripheral addresses are assigned to the SingleTact sensors using the address change Arduino project provided [53]. The Wire library also handles enabling the pull-up resistors.

ADC conversion happens by detecting the voltage at the designated pin and converting it to a digital binary value [54]. In the case of the Arduino Micro, a 10-bit resolution is provided [40] and can be used by assigning input and output pins with the **pinMode()** function [55]. In the case of the HC-SR04, using the **pulseIn()** function [56] will also assist when detecting pulses between 10 microseconds and 3 minutes in length. More detail with regards as to how these functions are used and their role in the microcontroller code is described in the microcontroller code section.

3.2 Microcontroller Code Development

The code for the functionalities of the Arduino Micro can be found in Appendix B1. The main sections will first be explained and then each individual function will be expanded upon.

3.2.1 Definitions and Variable initialisations

Lines 1 to 52 are the main definitions and global variable initialisations that are used throughout the project. Lines 1 and 2 include the main libraries used by the Micro which are "**Wire.h**" and "**USB-MIDI.h**". Line 4 creates a default MIDI USB instance for the Arduino Micro which allows for it to be used and viewed as a MIDI peripheral device. Lines 6 to 24 define certain values that are used as constants and the definitions provide for easier readability and debugging.

From Lines 27 to 52 the global variables are declared and initialised to zero. In the 51st and 52nd Lines function prototypes were listed due to errors given by the Arduino compiler regarding these functions.

3.2.2 Setup Section

The "**setup()**" section of the Arduino code is for initial setups and is only ran once in the entire program. At Line 55 the Arduino pins are setup so that they can communicate with the HC-SR04. The next 4 lines are to initialise the MIDI connection on channel 1, begin the I2C connection, and start a Serial connection of a 31250 samples/s baud rate [57].

3.2.2 Main Loop Section

In the "**loop()**" section is where the main program is run and is continuously looped. The first code that is executed is the "**measuredAndCCsend()**" function. This function reads all measurements from the connected sensors and sends the Control Change MIDI Messages using the SingleTact sensor data. This being outside of the main if-statement allows the TMC to also be used as a MIDI Effects controller.

If the measured distance is less than the defined maximum distance then the MIDI functionality is executed. Firstly, the idle state of the TMC is reset and the measured distance is entered to the "**buffer[]**" array. The array becomes populated at each iteration of the loop and the average is calculated each time. A weighted average was attempted but proved to be of no benefit so the raw data is not added to the final result and is instead multiplied by zero. The current note is then the current weighted average distance based on the output of the "**noteOut_Cmaj()**" function.

Lines 75 to 94 are where the MIDI messages regarding the notes are calculated and sent. If the current note is not the same as the previous note then the previous one is turned off, and the next one is sent. Lines 80 to 93 are not used in the final version of the project and are there to facilitate future work. At the end of the iteration, the previous note becomes the current note. This is in order to compare the different notes being played and to decide when to send a different note.

If the measured distance is more than the maximum distance, the idle counter is iterated to signify that the TMC is inactive. Once it has reached the value of 100, the TMC is reset. The pitch bend mode value is reset as well but this is not used in the final version of the product.

3.2.3 Functions: setupUSR()

In this function, the pinouts of the Arduino Micro for the HC-SR04 are specified. Using the definitions and the "**pinMode()**" function, pins number 18 and 19 are set to be INPUT and OUTPUT respectively.

3.2.4 Functions: getDistance()

At each iteration of the function, the variable to store the distance is set to 0. Using the HC-SR04 specification [33], the "**digitalWrite()**" function and a delay, the correct wave signal is sent in order for a received signal to occur. A short LOW pulse of 2 µs is given to the trigger pin and it is followed by a HIGH pulse lasting 10 µs, which is then set to low once that amount of time has passed. Using the provided equation in the specification and the "**pulseIn()**" [56] function, the distance of the object reflecting the ultrasonic wave is measured. It was found experimentally that the HC-SR04 rarely reached 0, so a distance offset is subtracted from the measured distance to remedy that. The measured distance value is then returned.

3.2.4 Functions: readDataFromSensor()

This function was referenced from a provided example Arduino file [58] and is used to read data from a SingleTact sensor using the I2C communications protocol. The addresses used are "0x06", "0x08" and "0x0A". More info as to how the SingleTact sensors handle I2C packets can be find in sections 2.3 and 2.4 of the SingleTact user manual [59].

3.2.4 Functions: testSensorOutputs()

Used when testing sensor data. Outputs are print in the Arduino IDE serial monitor. This will be showcased in the testing section of this report.

3.2.5 Functions: noteOut()

Using this function allows for the MIDI note values to be calculated and used in MIDI Note On or Note Off messages. The upper and lower bound of the range of detection for each note is calculated by equations (1) and (2),

$$LB = Range \times (Note\ Number - 1) \tag{1}$$

$$UB = Range\ \times Note\ Number - 1 \tag{2}$$

Where $Note\ Number$ is the values 1 through 13 to signify a full octave of notes to be played and $Range$ is the number defined as “range_cm” in the definitions. In the final project it was found experimentally that the optimal number is 4.

The notes outputted by this function are shown below in Table 6,

Table 6 MIDI Note Numbers and Notes outputted by the noteOut() function [49].

MIDI Note Number	Note
48	C4
49	C#4
50	D4
51	D#4
52	E4
53	F4
54	F#4
55	G4
56	G#4
57	A4
58	A#4
59	B4
60	C5

3.2.5 Functions: noteOut_Cmaj()

Similarly to the “**noteOut()**” function, this has the exact same functionality but it is tuned to play the C major scale instead of outputting all of the notes in an octave. This allows for a more Western and “in-tune” note selection. The note MIDI note numbers outputted are therefore: 48, 50, 52, 53, 55, 57, 59 and 60.

3.2.6 Functions: popBuffer()

When a distance value is measured, it is inputted in this function and gets added to the distance buffer array. The function works by duplicating the value in the current index starting from the final one, and moving it to the right by one. This allows for the first index of the array to be a duplicate of the second and could then therefore be replaced by the inputted distance value. An example of how this function works can be found in Figure 37.

$$[a\ b\ c\ d\ e]$$

$$[a\ b\ c\ d\ d]$$

$$[a\ b\ c\ c\ d]$$

$$[a\ b\ b\ c\ d]$$

$$[a\ a\ b\ c\ d]$$

$$[x\ a\ b\ c\ d]$$

Figure 37 Example functionality of the popBuffer() function with 5 entries. Each letter represents a number with x being the input of the new distance value.

3.2.6 Functions: calcAvg()

The average is calculated using this function. It is done by iterating the buffer array based on the desired buffer size and adding all of the values to a single variable. This variable is then divided by the buffer size to get the average.

3.2.6 Functions: reset()

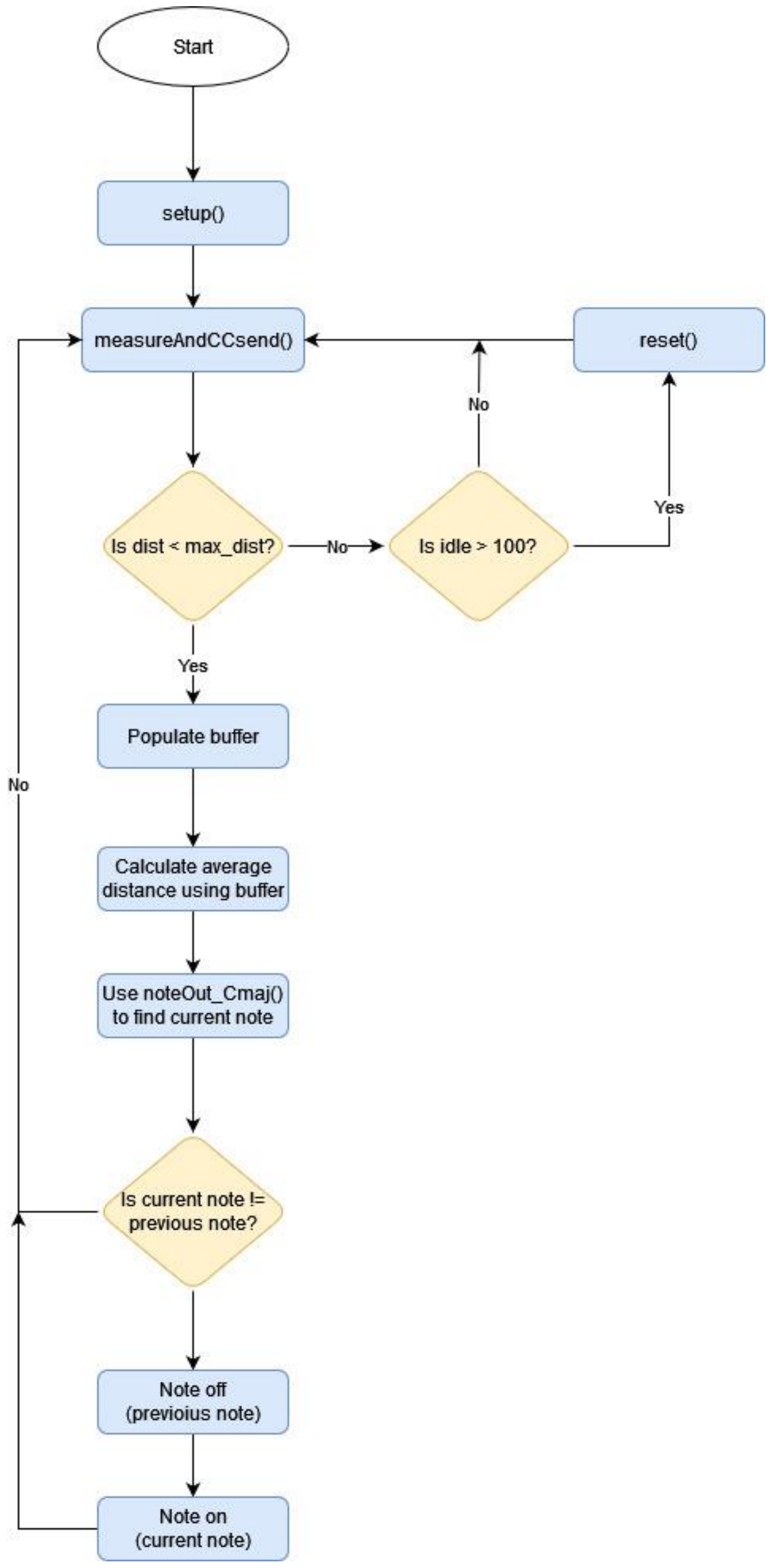
Resetting most of the required variables is done by this function. Firstly, it turns the TMC off, and then iterates each MIDI note number to turn the notes off using the "**sendNoteOff()**" function.

3.2.7 Functions: measureAndCCsend()

This function is the first one executed in the main loop. It firstly calls the “**getDistance()**” function and assigns its value to a variable. Then the value of each SingleTact sensor is read using the “**readDataFromSensor()**” function. Due to MIDI requiring event data to be between 0 and 127 the “**map()**” function is used [60] to map the received SIngleTact values. These values are between 256 or 260 and 1022 due to the sensors being uncalibrated. The minimum outputted value of the SingleTact sensors was observed experimentally per sensor and the minimum value of the function was set on based on it.

To minimise noise, the values above 4 are used to send MIDI Control Change (CC) data. CC data is sent using the “**MIDI.setControlChange()**” function at MIDI Channel 1. The CC values sent are 7, 12 and 13 as seen on Table 5. If the values are not more than 4, then the received data is set to be 0 in order to reset its value.

3.2.8 Flowchart



Start
setup()
measureAndCCsend()
reset()
No
Yes
Is dist < max_dist?
No
Is idle > 100?
Yes
Populate buffer
No
Calculate average distance using buffer
Use noteOut_Cmaj() to find current note
Is current note != previous note?
Note off (previoius note)
Note on (current note)

Figure 38 High level flowchart of microcontroller code. Rectangular boxes.

3.3 Application Development

As aforementioned, the connection will be to an Android phone, therefore Android Studio was used to develop the app. It is the recommended Android Integrated Development Environment for developing Android applications. Kotlin or Java were considered as viable programming languages but Java was preferred due to its longstanding documentation and variety of examples. Constraints exist with regards as to which Android API to use, due to MIDI being supported from Android API 23 (Marshmallow) and above. Android API 29 (Q) also added MIDI 2.0 functionalities, but Android API 23 is the minimum API that MIDI app development was feasible [61].

An example provided by the Android Developers [62] was used to aid the application development . A different Graphical User Interface (GUI) was created as well as a synthesiser using a sine wave instead of the default saw wave. Variables were also manipulated in specific ways as to allow the user to change the outputted sound using the SingleTact sensors. Any specific differences or additions to the original example files are commented with the word "ADDITIONS". Application code where the files were changed or modified is shown in Appendix B2. They are separated in classes for code and functionality organisation.

3.3.1 GUI

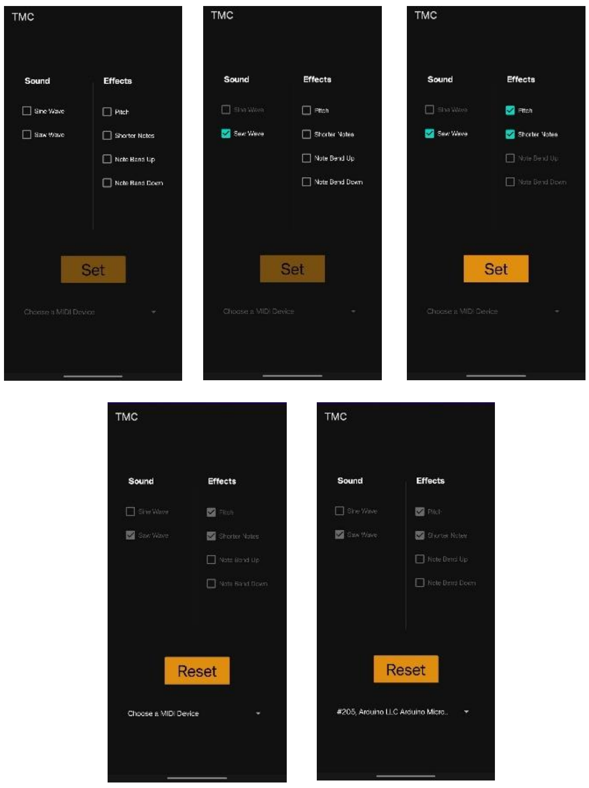


Figure 39 Five main states of the app. From top left to bottom right, initial state, sound selection state, effect selection state, set state, playing state.

The GUI was decided to be as minimalist as possible so that the user is not overwhelmed with options and the main features are highlighted. In Figure 39, the way the app adapts at the same time the user is interacting with it is shown. The application starts with all checkboxes unchecked and with the main button and spinner disabled. Once the user selects a sound and two effects, the button is made active in order for the user to press it and set the options they have selected. Once the button is pressed, the checkboxes become inactive, the text on the button changes to “Reset” and the spinner [63] is enabled. The user now can either reset and go back to the first state and re-check their desired options or they can select the Arduino from the spinner and begin playing with the TMC.

3.3.2 Classes: SelectionConfig()

This class was created to store user the configuration and it allows for better code and variable organisation. It is a static class which means it is only created once and the values are overwritten when the appropriate code is called. Variables are declared as “private static” and getters or setters are used to access or write to these variables. This class allows for the storing of a Sound ID variable, two Effect ID variables and a boolean Set variable.

These variables reflect the different interactions the user has with the app. It can be viewed in Appendix B2.1. The different options on the GUI represent different values within the ”SelectionConfig.java” class, as shown in Table 7.

Table 7 Different possible values of the variables in the SelectionConfig. java class. These change based on the selections made by the user.

On GUI	**In Application**
Sin Wave	sound_id = 1
Saw Wave	sound_id = 2
Pitch	effect_id = 1
Shorter Notes	effect_id = 2
Pitch Bend Up	effect_id = 3
Pitch Bend Down	effect_id = 4

3.3.2 Classes: MainActivity.java

Here is the main file where the project then branches off into different functions for different functionalities. It can be found in Appendix B2.2. The functionality described in the GUI is achieved using Lines 100 to 227 and Lines 230 to 321 in the MainActivity.java file. These are the functions that are executed when a checkbox is checked, or when the button is pressed.

3.3.2.1 Imports

Most imports were automatically generated by Android Studio when the code was being written. The imports for MIDI functionality and the Android Example files were manually imported.

3.3.2.2 Functions: First Execution and onCreate()

On first executions of this class, there are some variable initialisations and the “onCreate()” function is executed (Lines 24 to 59). This function sets up the initial states of the checkboxes, button, and spinner. It also checks if MIDI is supported by the device the application is being run on, and if not then MIDI is not setup and the application does nothing.

3.3.2.3 Functions: Android Example

Lines 62 to 96 are lines used from the Android Example provided. The “setupMIDI()” function initialises the MIDI connection by getting the device selected from the MIDI spinner and connecting to it. The “closeSynthResources()” and “onDestroy()” functions exist to allow the app to stop outputting gracefully and with no crashing. The “MyPortsConnectedListener()” function is used to check whether the connection to the selected device is successful or in error.

3.3.2.4 Functions: onCheckBoxClicked()

When a checkbox is clicked this function is executed. It goes through and checks the state of each checkbox in the GUI and reacts accordingly when a specific checkbox is checked. For example, in Lines 114 to 126, it checks which sound checkbox is selected and disables the other one.

Lines 128 to 213 are the same idea as the sound checkboxes but a lot more complicated since instead of one choice, the user makes two. For example, in the if-statement in Lines 130 to 141, a variety of checks occur. It checks if the checkbox is checked, and assigns the corresponding effect value. It also checks if the checkbox is deselected and the resets the effect value in the SelectionConfig class.

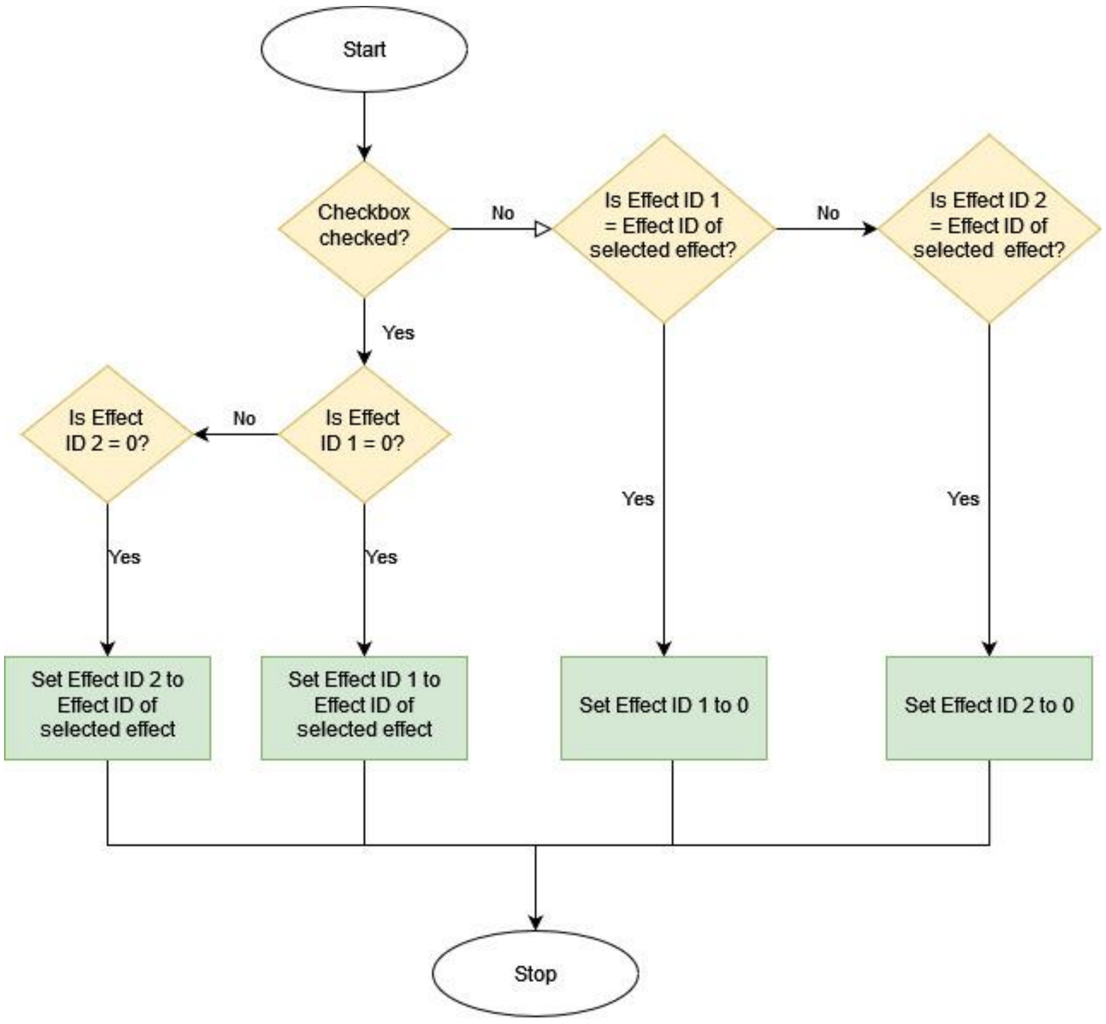

Start
Checkbox checked?
No
Is Effect ID 1 = Effect ID of selected effect?
No
Is Effect ID 2 = Effect ID of selected effect?
Yes
Is Effect ID 2 = 0?
No
Is Effect ID 1 = 0?
Yes
Yes
Yes
Yes
Set Effect ID 2 to Effect ID of selected effect
Set Effect ID 1 to Effect ID of selected effect
Set Effect ID 1 to 0
Set Effect ID 2 to 0
Stop

Figure 40 High level flowchart showing how the effect IDs are assigned. Rectangular boxes are actions and rhombus shapes are checks.

The next section of code in Lines 183 to 212 simply detects which checkboxes have been selected and disables the rest. The different combinations of options is shown in Table 4. This gives guidance to the user that no other selections should be done and that they can press the set button. To check if all selections are made the “effects_selected” and “sound_selected” variables are assigned to be true at the correct points in the code. If both are true then the user can set their configuration.

Table 8 Showing inactive checkboxes based on selections. Each number is the effect ID of the effect.

Selection 1	Selection 2	Inactive
1	2	3, 4
1	3	2, 4
1	4	2, 3
2	3	1, 4
2	4	1, 3
3	4	1, 2

3.3.2.5 Functions: onButtonClicked()

When the button is pressed this function is executed. Its purpose is to notify the program that no other selections are to be made and to also show to the user that no other selections are allowed. This is done by disabling and enabling the correct GUI components. Moreover, it is used to reset the program to its initial state, and let the user start over with new sound selections and new effects. Error checking also occurs where if the effects selected somehow have the same values, the program outputs an error.

3.3.3 Classes: EnvelopeADSR.java

The ADSR acronym in the name of this class stands for Attack, Decay, Sustain and Release and each term described affects that aspect of the specific sound [64]. This class was provided by the Android Example and only had some minor modifications made to it. In order for the “mSustainLevel” variable to be set and accessed by another class the “setSustainLevel()” function and the aforementioned variable are set to “static” (Lines 58 to 65 and Line 35 in Appendix B2.3). This allows for the Shorter Notes effect to take place.

3.3.4 Classes: SineVoice.java

The “SineVoice” class is used to create and access a sine wave synthesiser using the provided “SineOscillator” class. It is different to the provided saw wave synthesiser.

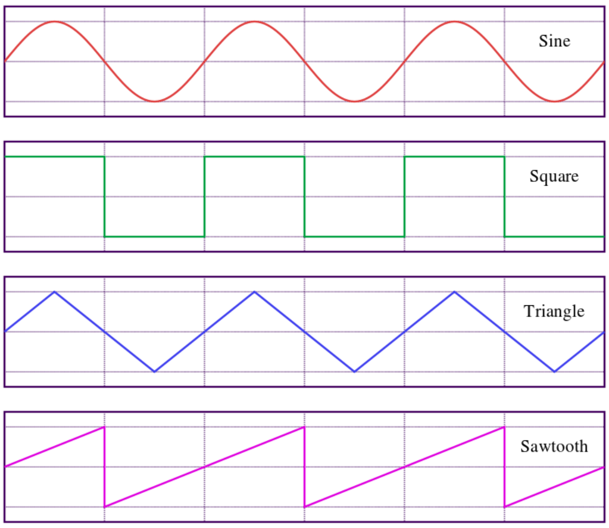


Figure 41 Examples of different wave shapes. On the top and bottom there is the Sine Waveform and the Saw/Sawtooth Waveform used in the project [63].

In Figure 41, the wave shapes of both sine and saw waves are shown. Each wave type requires a different oscillator and both types of oscillator classes are provided [65]. In the "SineVoice" class, the "SineOscillator" is used in the same way the "SawOscillator" is used in the "SawVoice" class. Code for the respective classes can be found in Appendix B2.4, B2.5, B2.6 and B2.7. A sine oscillator and envelope are instantiated and used to create the synthesiser using the same class methods. This class is used when the user selects the Sine Wave checkbox in the GUI.

3.3.5 Classes: SynthEngine.java

In "SynthEngine.java" is where MIDI and Synthesiser classes come together and all MIDI messages are received and executed. The first check that occurs is in Appendix B2.8, Lines 230 to 240 where based on the user's selection of sound and the received sound ID, the outputted wave is either a sine wave or a square wave. In Lines 96 to 119 the main switch-case statement occurs which checks what kind of MIDI event is being received. The cases that occur in this project are "MidiConstants.STATUS_NOTE_OFF", "MidiConstants.STATUS_NOTE_ON" and "MidiConstants.STATUS_CONTROL_CHANGE".

In the case of a Note Off or Note On MIDI message, it is handled by the corresponding functions provided by the Android Example. The addition to this switch-case statement is the case where a Control Change is sent by the TMC, in Lines 112 to 115. This handles the effect values on the outputted sound. As aforementioned in Table 5, the TMC sends Control Change numbers 7, 12 and 13. These can be mapped to any action through a switch-case statement, which is how it is done in the "controlChange()" function in Lines 271 to 319.

When the right-most SingleTact which is the Control Change (CC) number of 7, sends a MIDI message, it is reflected by the corresponding code in the case of 7. That CC number and in turn that SingleTact sensor is used to lower the volume of the outputted sound. This is done by changing the "velocity_cc" variable in Line 274 by subtracting the received CC value, and passing it to the "noteOn()" function in Line 101 of the switch-case statement.

The code for CC numbers 13 and 12 in Lines 276 to 319 is the same, so only one section will be examined. This is because the two SingleTacts send different CC numbers, but both must be able to interact with all of the effects. It checks which effect ID is selected and then acts accordingly based on Table 7. All CC values sent are between 0 and 127 to comply with the MIDI standard.

3.3.5.1 Pitch

The pitch effect is executed in Lines 277 to 282. The "mFrequencyScaler" variable is manipulated to change the scaling of the note's frequency. The value of this variable is passed to the "noteOn()" function in Line 259 and then the next note played will have the new value.

$$Frequency\ Scaler = 1 + \frac{CC\ Value}{16} \tag{3}$$

Equation (3) is used to calculate the frequency scaling. Due to the CC value being between 0 and 127 and only setting values above 10, the range of values for the Frequency Scaler will be 1.625 to 9. Since CC value is an integer (represented by the controlValue variable) it is truncated each time the calculation is executed.

Frequency scaling allows for a greater range for the TMC. When multiplying the frequency by an integer number, another note is played corresponding to that frequency [66]. For example when multiplying the note C2 by 2, the note C3 is reached and when multiplying by 3 then the note G3 is reached.

3.3.5.2 Shorter Notes

For the Shorter Notes effect, a function in the EnvelopeADSR class was set to “**public static**” in order to allow for access in the SynthEngine class. This was discussed in section 3.3.3. The value for the “**setSustainLevel()**” function in Line 286 is calculated and stored in Line 285 in the “**changeSustain**” variable.

$$Change\ Sustain\ = \frac{CC\ Value}{256} \tag{4}$$

The variable is changed using Equation (4) and with only accepting CC values above 10, it allows for values between 0.039 and 0.496. This range is acceptable due to the default value of 0.3.

3.3.5.3 Pitch Bend Up & Pitch Bend Down

Both of these effects use the same ideas and functions. The “**pitchBend()**” function is used for both. The “**map()**” function from the Arduino Reference [60] is also implemented into the application in Lines 321 to 324. It allows for easier calibration of the CC value inputs so that they are within MIDI standard.

The way pitch bending occurs within the app is uncommon within the MIDI standard, but shows the flexibility of the MIDI Control Change Messages. A pitch bend is executed in Lines 310 to 314 when the effect is selected from the GUI and a CC number of 12 or 13 occurs. In traditional MIDI messages, a separate Pitch Bend message exists with its own Event Value which is different to the method used in the app. However, in both cases the “**pitchBend()**” function is used for the pitch bend effect.

For the Pitch Bend Up effect, a value between 8192 and 16384 is required. For the Pitch Bend Down effect, a value between 0 and 8192 is needed instead. The value of 8192 means no pitch bend will happen to the current note [67].

This is different than from pitching up the note using frequency scaling in the Pitch effect. The pitch bend effect can reach notes that are not within any Western scale, whereas the Pitch effect simply multiplies the note frequency to change the note played.

3.3.8 AndroidManifest.xml

The Android Manifest file is an automatically generated file, but some manual additions were required in order for the application to work. In Appendix B2.9 Line 14 was changed to use the custom icon for the application. Lines 31 to 41 were required in order for the application to run and were referenced from the Android Example library.

3.3.9 Android Example Classes

From the referenced Android Example [62], the only files changed were EnvelopeADSR, SineVoice, SynthEngine, AndroidManifest, activity_main.xml (the GUI) and MainActivyty. The rest are as downloaded from the example. The same directory and file structure was used as to eliminate unnecessary file directory changes within the code.

3.3.10 Other Files

All files that are not discussed in this report are automatically generated or made with Android Studio Tools (eg. app icon) and therefore will not be mentioned.

3.4 Software Development Alternatives

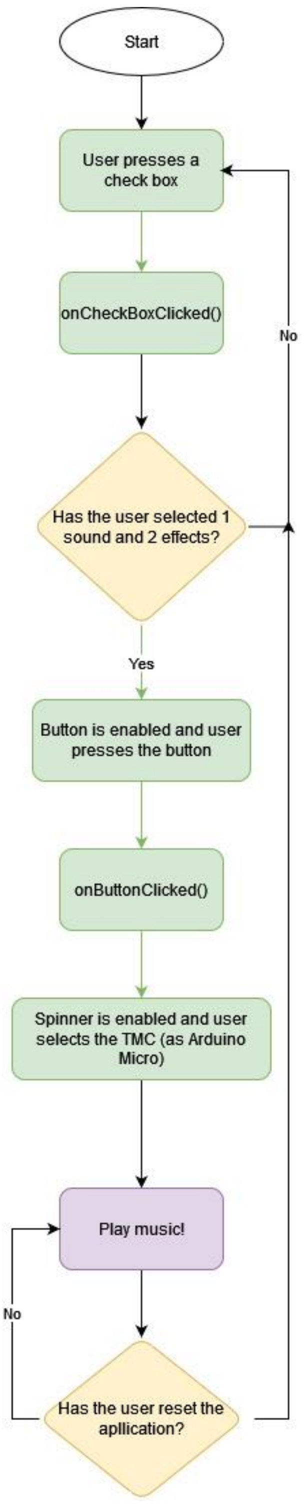
There are a variety of alternatives for application development. One alternative was using **MIT App Inventor** [68]**,** but more control was required for each function executed so a lower level approach was preferred. **JUCE** [69] could also have been used for creating an application with MIDI functionality, but due to the Android Example it was not chosen as the preferred method. Another example by **Mobileer** [70] could also have been used but it was noticed that it had nearly identical code to the Android Example. Therefore due to the example being provided by Android themselves, their example was preferred.

3.5 Software Development Issues

Multiple issues arose during software development. The two major ones were issues with the Arduino MIDI library and issues with application code execution. Initially, the library for USB and MIDI functionality used was **MIDIUSB** [71], but there were issues with detecting MIDI events on an Android device. Multiple tests with different USB cables, different MIDI devices, and different Android phones were conducted, with the conclusion being that the Arduino code was the issue. The library was then changed to the USB-MIDI library that is currently used in the project.

It was noted that when code was added or commented out, the app would not run the code. After getting some assistance with Java from an experienced Computer Science professor, it was discovered that there were duplicate files and that the project was falling back on those files when building. Furthermore, it was found that the app fell back on the Android Example application that was built, and ignored its own code. The app was then debugged with no other apps installed, which fixed the issue.

3.6 Flowchart



Start
User presses a check box
onCheckBoxClicked()
No
Has the user selected 1 sound and 2 effects?
Yes
Button is enabled and user presses the button
onButtonClicked()
Spinner is enabled and user selects the TMC (as Arduino Micro)
Play music!
No
Has the user reset the apllication?

Figure 42 High level flowchart of app functionality. Rectangular boxes are actions and rhombus shapes are checks.

4.0 Chapter IV – Testing and Results

4.1 Hardware Testing

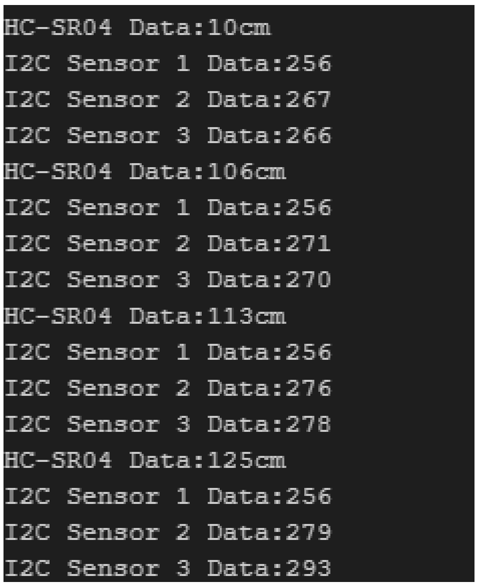
Hardware was tested using the Arduino IDE and checking the output. In the tests with test code H2 and H5, the "testSensorOutputs()" is being mentioned (Lines 174 to 190 in Appendix B1.1). The variables used are based on the component(s) being tested and the function is placed at Line 65.

Table 9 Hardware testing outcomes. Test code used to refer to the tests.

Test Code	Test	Method	Expected Outcome	Observed Outcome	Comments
H1	Arduino Micro being able to load code.	Connecting to Arduino IDE and outputting the string "test" on the Serial Monitor.	Outputted "test" on serial monitor.	Outputted "test" on serial monitor.	Test passed.
H2	HC-SR04 outputting a correct measured distance.	Using the "testSensorOutputs()" and "getDistance()" functions with the "dist" variable while checking the output on the Serial Monitor. A ruler was used to check distance measurement output and hand was positioned at 30 cm.	If hand placed at 30 cm then the value outputted should be 25 cm due to the offset.	25 cm on average but with noise values ranged from 23 to 28 cm.	Test passed. No averaging was used for this test.
H3	SingleTact sensor (individual).	Using the provided "SingleTactDemo.ino" [72] and reading the output on the Serial Monitor.	Values from 256 to 1024.	Values from 256 to 1022 and two sensors have a minimum value of 260.	Test passed. Range was adjusted in code so that all three have same outputs.

Table 10 Second table for hardware testing outcomes.

Test Code	Test	Method	Expected Outcome	Observed Outcome	Comments
H4	SingleTact sensor (all three).	Using the provided multi-sensor demo [73] and modifying it for three sensors instead of two.	Same results as the previous test, but all outputted at the same time on the Serial Monitor.	All readings outputted and inspected on serial monitor.	Test passed.
H5	All four sensors together can provide a measurement simultaneously.	Using all sensor measurement variables (Lines 26 to 30) and the "testSensorOutput()" function.	All previous sensor results outputted on the serial monitor at the same time.	All outputs on the Serial Monitor.	Test passed.

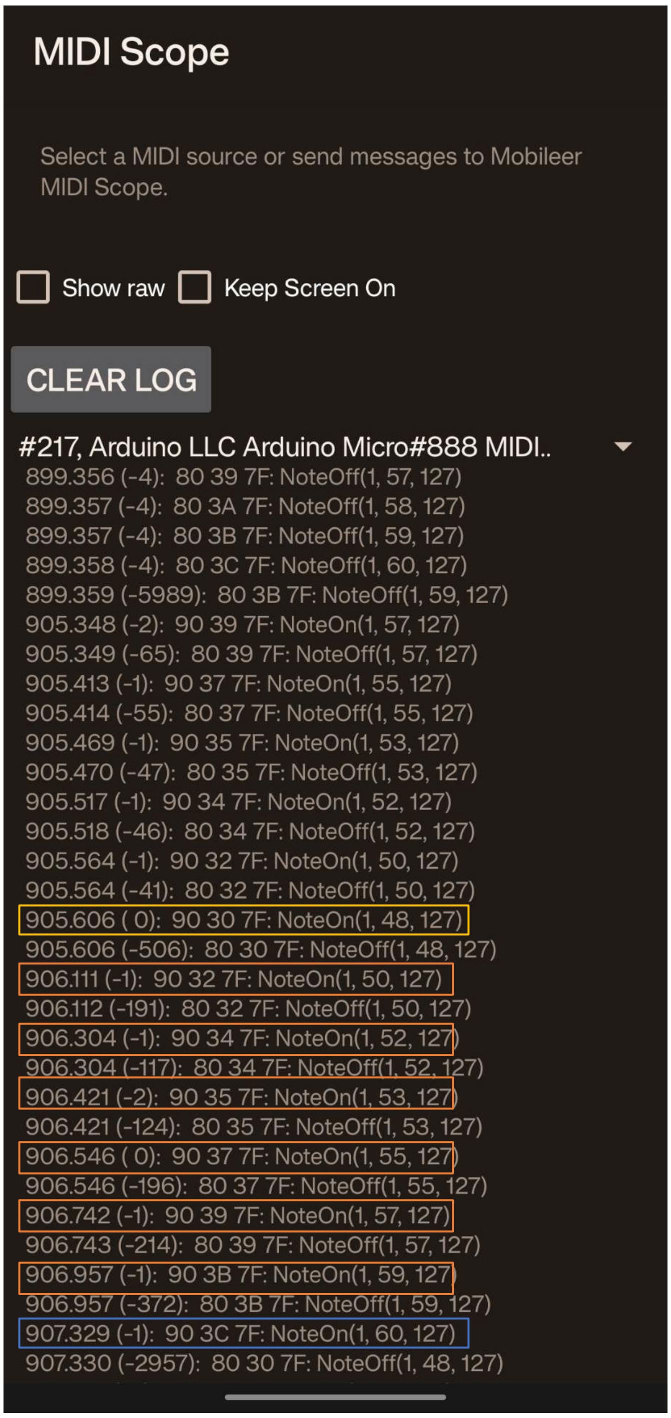


```
HC-SR04 Data:10cm
I2C Sensor 1 Data:256
I2C Sensor 2 Data:267
I2C Sensor 3 Data:266
HC-SR04 Data:106cm
I2C Sensor 1 Data:256
I2C Sensor 2 Data:271
I2C Sensor 3 Data:270
HC-SR04 Data:113cm
I2C Sensor 1 Data:256
I2C Sensor 2 Data:276
I2C Sensor 3 Data:278
HC-SR04 Data:125cm
I2C Sensor 1 Data:256
I2C Sensor 2 Data:279
I2C Sensor 3 Data:293
```

Figure 43 Output on Serial Monitor of Arduino IDE with all four sensors. Displays the output of H5 which also encompasses the H1 to H4 test codes

4.2 Software Testing

Testing for the software was done while connected to the TMC app (in Tests S1, S2, S7 and S8). The MIDI message detection on the PC was done using MidiView [74] and on Android it was done using MIDI Scope [75]. Notes with a number next to the note name (eg. C3 and C4) is the same note in different octaves. The different states of the app are listed and displayed in Figure 39.


MIDI Scope
Select a MIDI source or send messages to Mobileer MIDI Scope.
Show raw
Keep Screen On
CLEAR LOG
#217, Arduino LLC Arduino Micro#888 MIDI..

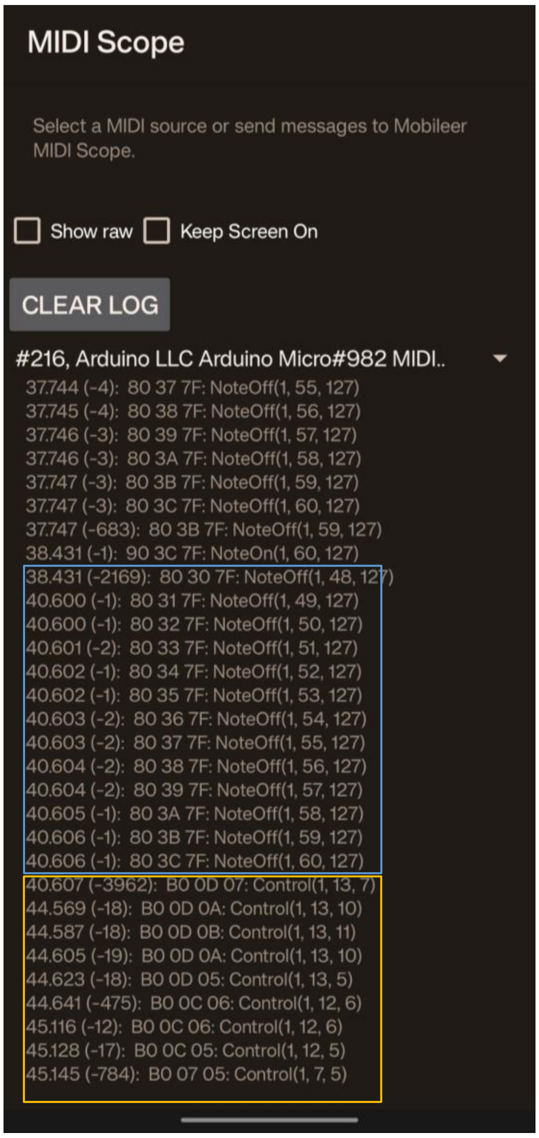
```
899.356 (-4):  80 39 7F: NoteOff(1, 57, 127)
899.357 (-4):  80 3A 7F: NoteOff(1, 58, 127)
899.357 (-4):  80 3B 7F: NoteOff(1, 59, 127)
899.358 (-4):  80 3C 7F: NoteOff(1, 60, 127)
899.359 (-5989):  80 3B 7F: NoteOff(1, 59, 127)
905.348 (-2):  90 39 7F: NoteOn(1, 57, 127)
905.349 (-65):  80 39 7F: NoteOff(1, 57, 127)
905.413 (-1):  90 37 7F: NoteOn(1, 55, 127)
905.414 (-55):  80 37 7F: NoteOff(1, 55, 127)
905.469 (-1):  90 35 7F: NoteOn(1, 53, 127)
905.470 (-47):  80 35 7F: NoteOff(1, 53, 127)
905.517 (-1):  90 34 7F: NoteOn(1, 52, 127)
905.518 (-46):  80 34 7F: NoteOff(1, 52, 127)
905.564 (-1):  90 32 7F: NoteOn(1, 50, 127)
905.564 (-41):  80 32 7F: NoteOff(1, 50, 127)
905.606 ( 0):  90 30 7F: NoteOn(1, 48, 127)
905.606 (-506):  80 30 7F: NoteOff(1, 48, 127)
906.111 (-1):  90 32 7F: NoteOn(1, 50, 127)
906.112 (-191):  80 32 7F: NoteOff(1, 50, 127)
906.304 (-1):  90 34 7F: NoteOn(1, 52, 127)
906.304 (-117):  80 34 7F: NoteOff(1, 52, 127)
906.421 (-2):  90 35 7F: NoteOn(1, 53, 127)
906.421 (-124):  80 35 7F: NoteOff(1, 53, 127)
906.546 ( 0):  90 37 7F: NoteOn(1, 55, 127)
906.546 (-196):  80 37 7F: NoteOff(1, 55, 127)
906.742 (-1):  90 39 7F: NoteOn(1, 57, 127)
906.743 (-214):  80 39 7F: NoteOff(1, 57, 127)
906.957 (-1):  90 3B 7F: NoteOn(1, 59, 127)
906.957 (-372):  80 3B 7F: NoteOff(1, 59, 127)
907.329 (-1):  90 3C 7F: NoteOn(1, 60, 127)
907.330 (-2957):  80 30 7F: NoteOff(1, 48, 127)
```

Figure 44 MIDI note output detection. Yellow box highlights the minimum note and blue highlights the maximum note. In orange are the rest of the notes outputted by the "noteOut_Cmaj()" function.

4.2.1 Microcontroller Code Testing

Table 11 Microcontroller code testing.

Test Code	Test	Method	Expected Outcome	Observed Outcome	Comments
S1	TMC stays idle when not in use.	Hand over the HC-SR04 for 5 seconds to get sensor measurements. Then removed and waited for TMC to be idle.	TMC shuts off.	TMC shuts off.	Test passed.
S2	TMC does not exceed maximum note or goes below the minimum note.	Using MidiView and looking at the maximum and minimum outputted notes.	Notes C3 and C4.	Note C3 and C4.	Test passed.
S3	MIDI Note On message sent.	Check on MidiView for the note on message.	MIDI Note On.	MIDI Note On.	Test passed.
S4	MIDI Note Off message sent.	Check on MidiView for the note off message.	MIDI Note Off.	MIDI Note Off.	Test passed.
S5	MIDI Control Change message (numbers 7, 12 and 13) sent independently of the distance measurements.	Check on MidiView for the different control change messages.	MIDI Control Change with values 7, 12 and 13.	MIDI Control Change with values 7, 12 and 13.	Test passed.
S6	All MIDI messages can be observed on an Android device.	Check on MIDI Scope for all the aforementioned messages.	All MIDI messages used are observed.	All MIDI messages used are observed.	Test passed.


MIDI Scope

Select a MIDI source or send messages to Mobileer MIDI Scope.

```
#216, Arduino LLC Arduino Micro#982 MIDI..
37.744 (-4):  80 37 7F: NoteOff(1, 55, 127)
37.745 (-4):  80 38 7F: NoteOff(1, 56, 127)
37.746 (-3):  80 39 7F: NoteOff(1, 57, 127)
37.746 (-3):  80 3A 7F: NoteOff(1, 58, 127)
37.747 (-3):  80 3B 7F: NoteOff(1, 59, 127)
37.747 (-3):  80 3C 7F: NoteOff(1, 60, 127)
37.747 (-683):  80 3B 7F: NoteOff(1, 59, 127)
38.431 (-1):  90 3C 7F: NoteOn(1, 60, 127)
38.431 (-2169):  80 30 7F: NoteOff(1, 48, 127)
40.600 (-1):  80 31 7F: NoteOff(1, 49, 127)
40.600 (-1):  80 32 7F: NoteOff(1, 50, 127)
40.601 (-2):  80 33 7F: NoteOff(1, 51, 127)
40.602 (-1):  80 34 7F: NoteOff(1, 52, 127)
40.602 (-1):  80 35 7F: NoteOff(1, 53, 127)
40.603 (-2):  80 36 7F: NoteOff(1, 54, 127)
40.603 (-2):  80 37 7F: NoteOff(1, 55, 127)
40.604 (-2):  80 38 7F: NoteOff(1, 56, 127)
40.604 (-2):  80 39 7F: NoteOff(1, 57, 127)
40.605 (-1):  80 3A 7F: NoteOff(1, 58, 127)
40.606 (-1):  80 3B 7F: NoteOff(1, 59, 127)
40.606 (-1):  80 3C 7F: NoteOff(1, 60, 127)
40.607 (-3962):  B0 0D 07: Control(1, 13, 7)
44.569 (-18):  B0 0D 0A: Control(1, 13, 10)
44.587 (-18):  B0 0D 0B: Control(1, 13, 11)
44.605 (-19):  B0 0D 0A: Control(1, 13, 10)
44.623 (-18):  B0 0D 05: Control(1, 13, 5)
44.641 (-475):  B0 0C 06: Control(1, 12, 6)
45.116 (-12):  B0 0C 06: Control(1, 12, 6)
45.128 (-17):  B0 0C 05: Control(1, 12, 5)
45.145 (-784):  B0 07 05: Control(1, 7, 5)
```

Figure 45 MIDI outputs of the CC messages. Highlighted in yellow are the 7, 12 and 13 MIDI CC numbers sent. In blue is the code executed when the TMC turns off and is in idle state.

Figure 44 shows the outputs of the "noteOut()" function (Appendix B1.1 Lines 222 to 240) which has the output discussed in Section 3.2.5. Therefore, by interpreting the results in Figure 44 and Figure 45, results of tests with test code S1 to S6 are shown.

4.2.2 Application Testing

Table 12 Application testing outcomes.

Test Code	Test	Method	Expected Outcome	Observed Outcome	Comments
A1	App starts at the correct initial state	Open the app	Initial State	Initial State	Test passed.
A2	Correct check boxes become disabled based on the configuration	Using Table 8, and checking which ones are greyed out.	The entries in Table 8 are true.	The entries in Table 8 are true.	Test passed.
A3	"Set" button becomes enabled when all choices are made	Select one sound and two effects.	The button is enabled.	The button is enabled.	Test passed.
A4	"Reset" resets the choices	Pressing the reset button once in the set or playing states.	App completely resets.	App completely resets.	Test passed.
A5	The spinner correctly connects and disconnects a MIDI device	Selecting and deselecting the MIDI device on the spinner and observing the outputted message on the GUI.	App correctly connects and disconnects the TMC when selected. "Device Connected" appears as a Toast [76] output.	App correctly connects and disconnects the TMC when selected. "Device connected" appears.	Test passed.

Table 13 Second table for application testing outcomes.

Test Code	Test	Method	Expected Outcome	Observed Outcome	Comments
A6	App outputs a note for each option possibility.	Use all possible options in Table 8 for both Saw and Sine wave configurations.	App interacts with all effects and configurations.	App interacts with all effects and configurations.	Test passed.
A7	Right-most sensor turns down the volume of the note played	Press sensor while playing with the TMC.	Volume turns down.	Volume turns down.	Test passed.

Outcomes of tests A1 to A5 are shown in “tests A1 to A5.mp4” either in the provided supplemental material or on the GitHub page under the “Images and Videos” folder. The files “saw wave outputs.mp4” and “sin wave outputs.mp4” show the outcomes of tests A6 and A7, also found in the same folder.

4.3 Results

Video files mentioned are provided in the "Images and Videos" folder.

Table 14 Results for the project objectives. Objective code for easier reference.

Objective Code	Objective Description	Result
O1	Theremin hardware interface using an ultrasonic transmitter and receiver.	Figure 43 shows some outputs of the HC-SR04 for hand measurements. In video "**singletact and app interaction results.MOV**" the right-most SingleTact is used for the volume change.
O2	Make the instrument able to send MIDI commands in order for it to be used in other applications.	Figure 44 and Figure 45 show the listed MIDI commands in Table 5 being sent and detected. In the video named "**DAW MIDI compatibility.mp4**", the versatility of the TMC is showed by mapping the SingleTacts to a filter of a digital synthesiser in a DAW. Achieved using FL Studio [77].
O3	Application created to provide phone interaction with instrument.	The files "**saw wave outputs.mp4**" and "**sin wave outputs.mp4**" show how the app interacts with the inputted instrument data.
O4	Implement three SingleTact sensors, with two being for effects.	Interaction of all three SIngleTact sensors (two for effects and one for volume) is shown in the video file named "**singletact and app interaction results.MOV**".
O5	Ability to play a simple song like "Amazing Grace".	Files named "**amazing grace take X.mp4**" where X is numbers 1 to 5, show different takes of this result.

5.0 Chapter V – Evaluation

5.1 Discussion

In Table 14, the different results for the different objectives are shown. For objective with code O1, the final product can behave as a Theremin with the HC-SR04 transceiver/receiver measuring the hand distances. The volume change is achieved with the right-most SingleTact being mapped solely to the volume of the next note. This fulfils the theremin requirements, however the chosen ultrasonic sensor is not very accurate with measuring hand distances due to the hands being a soft, irregular surface and the noise from the wave reflections in the room. To remedy this an averaging method was used as discussed in Section 2.4, but this removed the ability to pinpoint specific notes in the range of the TMC. For example, in the final product the ability to instantly play a note in any part of the scale is not possible, because the averaging method used to remove the noise, forced the product to always work in a sequential manner. This does remove some objectively musical capabilities of the TMC, but due to the subjective experience of music, it can be considered a unique feature of the product. Even though objective O5 was completed, it had difficulties being fulfilled because of this reason as well. The TMC can reach all of the required notes, but practice was required to compensate for the inaccuracies of the HC-SR04, making the project a difficult instrument to learn.

For objective O2, the TMC can be mapped to different controls in a DAW as shown, so it can be used effectively as a General MIDI controller. Other applications such as the ones used in testing, as well as the companion app, can also detect the MIDI commands being sent so it is compatible with all MIDI compatible applications. The SingleTacts can also be used to send MIDI CC messages (numbers 7, 12 and 13) while no notes are being played, therefore the TMC can also be used as a MIDI effect controller while another controller is being played. For example, the SingleTacts can be mapped to a virtual knob to change a certain value which can be controlled by another person separately. Another person could then be playing a keyboard MIDI controller and both of them could play music together by each one changing what the other hears using their controllers. An issue with the current set of commands being sent is that they are mapped for specific use in the app, so other for other apps would have to map the current MIDI Control Change commands to other actions. If the app connected to the TMC has no mapping compatibilities, the SingleTacts may have unpredictable behaviour such as changing random options or no effect at all. This depends on the individual app being examined.

The application created fully satisfies O3 objective. The app is a fully customised application made specifically for the TMC. It allows for customising of the sound output as well as which effects can be used at a time on the corresponding sound. Other MIDI controllers can be used with the app due to it being MIDI compatible, but their function depends on which MIDI commands they are able to send. If they can send the commands in Table 5, then the app can fully be used with no issues with such a MIDI controller.

The three SingleTact sensors are fully integrated with the TMC allowing for effect manipulation, assignment as well as volume change of the notes played. This fulfils objective O4, however the “Pitch” and “Shorter Notes” effects are only executed at the next note being played. This is also true for the SingleTact sensor assigned to the note volume. Minor confusion can occur while playing, since it is more common to want to affect what is currently playing rather than the next note the player has in mind. With practice this can be compensated for, allowing for smoother note transitions while using the volume sensor as well as the two affected effects.

The general playability of the TMC is an interesting point of discussion. The theremin is notoriously a difficult instrument to start playing ([78] [79] [80]) and to become proficient in. This is mainly due to there not being any instant feedback with which note the user is playing. On a guitar, a player can easily tell what note is being played based on the fret, but since the theremin is a fretless instrument, it is very easy to play out of tune notes and get lost. This is the same case with the TMC since it uses the same concepts as the theremin for music creation. One hand creates the pitch, and the other can control the volume or other effects. The same difficulty level applies to the TMC although in a different way due to it being tuned to the C major scale. The user requires practice to find where the different notes are placed vertically on top of the TMC but there is no need to have a near-pitch perfect ear to find the notes. The user also has to get used to the way notes are outputted sequentially and with the latency required to calculate the average each time. Coordination of the hands is also still required to use the effects and volume in a musical manner. Therefore, the TMC can be considered an introductory and beginner instrument to the theremin family due to it being tuned, but still having most of the theremin characteristics. It is playable the same ways as a theremin since it has very similar concepts and behaviour.

The TMC along with its companion app is very versatile once closely examined. It can output two types of sound wave shapes (sine and saw) and use four effects for each one. The effects provide a range spanning multiple octaves due to the “Pitch” effect. The “Note Bend Up” and “Note Bend Down” effects allow for it to reach out of tune notes, making the instrument sound more interesting and less robotic. The “Shorter Notes” effect allow for the TMC to be used in a more subtle way and when using the “Pitch” effect at the same time, it can provide some entertaining sounds. The type of music outputted by the TMC can be used in all kinds of music especially electronic music. It can provide a unique voice never heard before due to its one-of-a-kind interface and effect selection. Ambient music has historically used theremins for a more sci-fi texture, thus making the TMC useable for that as well. Due to DAWs and the TMC being MIDI compatible, the sounds can be edited and manipulated in unimaginable ways thus making the final product nearly limitless.

Portability is definitely an advantage of the TMC over other theremins on the market. Since it is a MIDI Controller, very few analogue or mechanical components are required in order for it to function thus making it portable and useable in a variety of spaces. The casing and the sensor cover also allow protection of the components, but the plastic is somewhat weak and can brake when under heavy weight. Sensor cover can however be lost since it is not attached to the casing in any way.

Safety is a concern for a final product out on the market. The TMC has some security hazards such as the top lid being opened with some force. This does allow for appreciation of the inner workings, but the wires and solder joints are exposed to being accidentally touched. Even though the maximum current and voltage (50 mA and 5 V) of the Arduino Micro is not lethal or harmful, it should be considered as an issue.

5.2 Conclusion

To conclude, this report described the creation of a phone-based tactile musical instrument. A theremin hardware interface was created using an HC-SR04 ultrasonic receiver/transmitter to measure the user's hand movements, as well as a SingleTact sensor used to change the volume. The final product can send MIDI messages which can be universally interpreted by MIDI compatible apps and DAWs. The MIDI compatible companion application created can also be used to interact with the TMC and change the sound output by modifying the sound and effect settings. Two of the three SingleTacts can be used for effects, with each being mapped to one of the four effects provided by the companion application. Using the application or any other MIDI app, the TMC can play a simple melody like "Amazing Grace" with some practice so that the mechanics of the product become more familiar.

The project started with initial market research to find gaps that could be used to inspire new solutions. The Theremin was observed to have not been used in the phone-based musical instrument market, and with also observing the popularity of MIDI controllers - a Theremin MIDI Controller was proposed. Based on that, components and concepts were thought out and therefore materialised as the project continued. Four 3D prints were made before deciding on the final one with another one being planned, but other project objectives took priority. An Arduino Micro was used with three SingleTact sensors and an HC-SR04 ultrasonic transmitter/receiver to accomplish the proposed project. MIDI functionality was accomplished using an Arduino library. The companion app was developed using the Java language and modified examples provided by the Android Developers. A custom GUI was created to allow unique interaction between the capabilities of the app and the user. Difficulties with 3D CAD design, Java coding and MIDI library compatibilities were the main obstacles for the project, due to lack of knowledge or experience. They were delt with as they occurred and possible alternatives were planned at each stage if there was no way around them. The final product from this project is a unique take on the MIDI Controller that does not exist on the market currently. It has near limitless creative possibilities and applications, while also completing all listed objectives. Therefore, it safe to conclude that the project is a success.

5.3 Further Work

A variety of features could be added on if extra time was provided but other project objectives took priority. A separate section of the app could be used to allow for customisation of the effects. For example, the pitch bend effect could be set to bend only half an octave. On the main app page or on a separate page, the current note played could also be displayed to aid the player visually. More effects could have been implemented by manipulating the oscillators provided, as well as create new ones such as a square wave to provide more customisability.

Currently in the microcontroller code, there exists the "pitchBend_mode" and "pitchBend_val" variables, as well as Lines 80 to 93 in Appendix B1.1. These when executed provide a different sounding experience that mimics a real life theremin. However, due to not allowing the user to change this value, the lines mentioned are not used. Therefore, one addition would have been a switch which would connect to the Arduino Micro ADC and ground pins, and would allow the user to select different modes for the TMC.

Some additions to the casing would be an indicator LED to provide confidence to the user that the TMC is working as it should. A screw could also have been put in place on the top lid to further prevent unwanted access to the wiring and for added safety. More external additions would have been a guide to allow the user to practice more easily. It could have been a piece of cardboard with note markings and it could be stood next to the TMC while playing.

References

[1] ‘What Is MIDI? How To Use the Most Powerful Tool in Music’, *LANDR Blog*, Jan. 13, 2020. https://blog.landr.com/what-is-midi/ (accessed Mar. 11, 2023).

[2] ‘What Is a DAW? A Guide to Digital Audio Workstations - 2023 - MasterClass’. https://www.masterclass.com/articles/what-is-a-daw (accessed Mar. 11, 2023).

[3] Keerthi, ‘15 Types of Musical Instruments – History and Categories with Images’, *Styles At Life*, Jan. 04, 2022. https://stylesatlife.com/articles/musical-instruments-types-with-names-and-pics/ (accessed Mar. 11, 2023).

[4] ‘What is a theremin? All you need to know about the quirky instrument’, *Classical Music*. https://www.classical-music.com/features/instruments/what-is-a-theremin/ (accessed Mar. 11, 2023).

[5] dhawan.sunidhi10, ‘Make a Simple Ultrasonic Theremin’, *Instructables*. https://www.instructables.com/Ultrasonic-Theremin-2/ (accessed Mar. 11, 2023).

[6] J. Nugent, ‘What is a MIDI controller/keyboard, what it does, and why you need one?’, *Higher Hz*, Jan. 21, 2021. https://higherhz.com/what-is-midi-controller-keyboard/ (accessed Mar. 11, 2023).

[7] ‘Italian musical terms’, *Musicca*. https://www.musicca.com/musical-terms (accessed Mar. 11, 2023).

[8] A. Aciman, ‘Meet the unassuming drum machine that changed music forever’, *Vox*, Apr. 16, 2018. https://www.vox.com/culture/2018/4/16/16615352/akai-mpc-music-history-impact (accessed Mar. 11, 2023).

[9] ‘IK Multimedia - iRig Keys 2’. https://www.ikmultimedia.com/products/irigkeys2/ (accessed Mar. 11, 2023).

[10] ‘IK Multimedia - iRig Keys I/O’. https://www.ikmultimedia.com/products/irigkeysio/index.php (accessed Mar. 11, 2023).

[11] ‘CME - Mobile Music Instruments | Always. One Step. Ahead.’, *xKeyAir*. https://xkeyair.com/xkey-air/ (accessed Mar. 11, 2023).

[12] ‘Linnstrument’. https://www.rogerlinndesign.com/linnstrument (accessed Mar. 11, 2023).

[13] ‘MPK Mini mk3 MIDI Controller | Akai Pro’. https://www.akaipro.com/mpk-mini-mk3 (accessed Mar. 11, 2023).

[14] ‘Novation Launchkey Mini MK3’, *Musikhaus Thomann*. https://www.thomann.de/gb/novation_launchkey_mini_mk3.htm (accessed Mar. 11, 2023).

[15] ‘Thomann Accessories for Mobile-Devices’, *Musikhaus Thomann*. https://www.thomann.de/gb/accessories_for_mobile-devices.html (accessed Mar. 11, 2023).

[16] ‘IK Multimedia - iRig Pro I/O’. https://www.ikmultimedia.com/products/irigproio/ (accessed Mar. 11, 2023).

[17] ‘IK Multimedia - iRig Stream’. https://www.ikmultimedia.com/products/irigstream/ (accessed Mar. 11, 2023).

[18] ‘Wireless Mixer - MIDI – Apps on Google Play’. https://play.google.com/store/apps/details?id=com.bti.wirelessMixer&hl=en_GB (accessed Mar. 11, 2023).

[19] ‘TouchDAW – Apps on Google Play’. https://play.google.com/store/apps/details?id=de.humatic.tdaw&hl=en_GB (accessed Mar. 11, 2023).

[20] ‘Midi Commander - Apps on Google Play’. https://play.google.com/store/apps/details?id=it.bordero.midicontroller&hl=en_US (accessed Mar. 11, 2023).

[21] 'MIDI Keyboard – Apps on Google Play'. https://play.google.com/store/apps/details?id=com.dreamhoundstudios.keyboard&hl=en_GB&gl=US (accessed Mar. 11, 2023).
[22] 'FluidSynth MIDI Synthesizer – Apps on Google Play'. https://play.google.com/store/apps/details?id=net.volcanomobile.fluidsynthmidi&hl=en_GB&gl=US (accessed Mar. 11, 2023).
[23] 'FL STUDIO MOBILE – Apps on Google Play'. https://play.google.com/store/apps/details?id=com.imageline.FLM&hl=en_GB&gl=US (accessed Mar. 11, 2023).
[24] 'Cubasis 3 - DAW & Studio App – Apps on Google Play'. https://play.google.com/store/apps/details?id=com.steinberg.cubasis3&hl=en_GB&gl=US (accessed Mar. 11, 2023).
[25] 'SampleTank 4 - The ultimate sound and groove workstation'. https://www.ikmultimedia.com/products/st4/index.php?p=info (accessed Mar. 11, 2023).
[26] 'LUMI Keys & App: The world's first all-in-one platform for learning music at home | LUMI'. https://playlumi.com/uk (accessed Mar. 11, 2023).
[27] 'Joué Play - Experiment Music Creation Freely – Joué Music Instruments'. https://jouemusic.com/en/collections/accessoires/products/joue-play?variant=40650305896646 (accessed Mar. 11, 2023).
[28] 'Inventory: 14 Movies From Two Ages Of Theremin Music'. https://www.avclub.com/inventory-14-movies-from-two-ages-of-theremin-music-1798209672 (accessed Mar. 11, 2023).
[29] 'Watch Jimmy Page Rock the Theremin, the Early Soviet Electronic Instrument, in Some Hypnotic Live Performances | Open Culture'. https://www.openculture.com/2016/09/jimmy-page-rocks-the-theremin.html (accessed Mar. 11, 2023).
[30] 'Doepfer A-178 Theremin', *Musikhaus Thomann*. https://www.thomann.de/gb/doepfer_a178.htm (accessed Mar. 11, 2023).
[31] 'Moog Theremini', *Musikhaus Thomann*. https://www.thomann.de/gb/moog_theremini.htm (accessed Mar. 11, 2023).
[32] 'Moog Claravox Centennial Theremin', *Musikhaus Thomann*. https://www.thomann.de/gb/moog_claravox_centennial_theremin.htm (accessed Mar. 11, 2023).
[33] '46130 | Ultrasonic Distance Sensor HC-SR04 5V Version | RS'. https://uk.rs-online.com/web/p/bbc-micro-bit-add-ons/2153181 (accessed Mar. 11, 2023).
[34] 'Micro USB Plug to USB-C Socket Adaptor, White - PSG91742'. https://cpc.farnell.com/pro-signal/psg91742/usb-c-female-micro-b-male-adapter/dp/CS33860 (accessed Mar. 11, 2023).
[35] 'USB 2.0 Type-C to USB Type-C Cable, 1m White - PSG91490'. https://cpc.farnell.com/pro-signal/psg91490/lead-usb2-0-type-c-type-c-white/dp/CS31098 (accessed Mar. 11, 2023).
[36] 'A000053 | Arduino, Micro Development Board | RS'. https://uk.rs-online.com/web/p/arduino/7717667 (accessed Mar. 20, 2023).
[37] 'MSP430F5500 data sheet, product information and support | TI.com'. https://www.ti.com/product/MSP430F5500 (accessed Mar. 11, 2023).
[38] 'FT312D-32L', *FTDI*. https://ftdichip.com/products/ft312d-32l/ (accessed Mar. 11, 2023).
[39] 'SingleTact_Spec-Sheet_V7.1.pdf'. https://www.singletact.com/storage/SingleTact_Spec-Sheet_V7.1.pdf (accessed Mar. 11, 2023).

[40] 'Arduino Micro', *Arduino Official Store*. https://store.arduino.cc/products/arduino-micro (accessed Mar. 11, 2023).
[41] 'Arduino Macro 3D Model'. https://grabcad.com/library/arduino-micro-1 (accessed Mar. 11, 2023).
[42] 'HC-SR04 3D Model'. https://grabcad.com/library/hc-sr04-ultrasonic-sensor-8 (accessed Mar. 11, 2023).
[43] Thingiverse.com, 'HC-SR04 Ultrasonic Rangefinder Mounting Bracket by Matter153'. https://www.thingiverse.com/thing:936318 (accessed Mar. 12, 2023).
[44] 'Devantech SRF08 Ultrasonic Range Finder - RobotShop'. https://uk.robotshop.com/products/devantech-ultrasonic-range-finder-srf08 (accessed Mar. 11, 2023).
[45] 'Arduino Uno Rev3', *Arduino Official Store*. https://store.arduino.cc/products/arduino-uno-rev3 (accessed Mar. 11, 2023).
[46] 'Arduino Due', *Arduino Official Store*. https://store.arduino.cc/products/arduino-due (accessed Mar. 11, 2023).
[47] lathoub, 'Arduino USB-MIDI Transport'. Mar. 06, 2023. Accessed: Mar. 14, 2023. [Online]. Available: https://github.com/lathoub/Arduino-USBMIDI
[48] 'Meta Events'. https://www.mixagesoftware.com/en/midikit/help/HTML/midi_events.html (accessed Mar. 14, 2023).
[49] 'MIDI note numbers and center frequencies | Inspired Acoustics'. https://www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies (accessed Mar. 14, 2023).
[50] 'Controllers'. https://www.mixagesoftware.com/en/midikit/help/HTML/controllers.html (accessed Mar. 14, 2023).
[51] 'I2C - SparkFun Learn'. https://learn.sparkfun.com/tutorials/i2c/all (accessed Mar. 16, 2023).
[52] 'Wire - Arduino Reference'. https://www.arduino.cc/reference/en/language/functions/communication/wire/ (accessed Mar. 14, 2023).
[53] 'StandaloneArduino/SingleTactAddressChange.ino at master · SingleTact/StandaloneArduino'. https://github.com/SingleTact/StandaloneArduino/blob/master/SingleTactAddressChange.ino (accessed Mar. 16, 2023).
[54] W. Storr, 'Analogue to Digital Converter (ADC) Basics', *Basic Electronics Tutorials*, Sep. 21, 2020. https://www.electronics-tutorials.ws/combination/analogue-to-digital-converter.html (accessed Mar. 16, 2023).
[55] 'pinMode() - Arduino Reference'. https://www.arduino.cc/reference/en/language/functions/digital-io/pinmode/ (accessed Mar. 16, 2023).
[56] 'pulseIn() - Arduino Reference'. https://www.arduino.cc/reference/en/language/functions/advanced-io/pulsein/ (accessed Mar. 16, 2023).
[57] '22. Midi - Sound Engineer's Pocket Book, 2nd Edition [Book]'. https://www.oreilly.com/library/view/sound-engineers-pocket/9780240516127/Chapter22.html (accessed Mar. 17, 2023).
[58] 'StandaloneArduino'. SingleTact, Mar. 19, 2022. Accessed: Mar. 17, 2023. [Online]. Available: https://github.com/SingleTact/StandaloneArduino/blob/ba3b8783b206c9b262a0fe49af55cc92f76caeca/SingleTactMultiSensorDemo.ino

[59] 'Single Tact Manual'. Accessed: Mar. 17, 2023. [Online]. Available: https://www.singletact.com/SingleTact_Manual.pdf
[60] 'Arduino Reference. map()'. https://cdn.arduino.cc/reference/en/language/functions/math/map/ (accessed Mar. 18, 2023).
[61] 'Developing MIDI applications on Android', *The MIDI Association*. https://www.midi.org/midi-articles/developing-midi-applications-on-android (accessed Mar. 11, 2023).
[62] 'media-samples/MidiSynth at main · android/media-samples', *GitHub*. https://github.com/android/media-samples (accessed Mar. 18, 2023).
[63] 'Spinners', *Android Developers*. https://developer.android.com/develop/ui/views/components/spinner (accessed Mar. 24, 2023).
[64] 'Attack, decay, sustain, and release', *Apple Support*. https://support.apple.com/en-gb/guide/logicpro/lgsife419620/mac (accessed Mar. 18, 2023).
[65] 'Wave Generators - Quorum Programming Language'. https://quorumlanguage.com/tutorials/dsp/audiowavegenerator.html (accessed Mar. 19, 2023).
[66] 'Frequencies of Musical Notes, A4 = 440 Hz'. https://pages.mtu.edu/~suits/notefreqs.html (accessed Mar. 20, 2023).
[67] C. Dobrian, 'Managing MIDI pitchbend messages | Computer Audio and Music Programming – 2014', Apr. 30, 2014. https://sites.uci.edu/camp2014/2014/04/30/managing-midi-pitchbend-messages/ (accessed Mar. 20, 2023).
[68] 'MIT App Inventor'. https://appinventor.mit.edu/ (accessed Mar. 20, 2023).
[69] 'JUCE: Tutorial: Handling MIDI events'. https://docs.juce.com/master/tutorial_handling_midi_events.html (accessed Mar. 20, 2023).
[70] P. Burk, 'android-midisuite'. Mar. 10, 2023. Accessed: Mar. 20, 2023. [Online]. Available: https://github.com/philburk/android-midisuite
[71] 'MIDIUSB Library for Arduino'. Arduino Libraries, Mar. 19, 2023. Accessed: Mar. 20, 2023. [Online]. Available: https://github.com/arduino-libraries/MIDIUSB
[72] 'StandaloneArduino/SingleTactDemo.ino at master · SingleTact/StandaloneArduino'. https://github.com/SingleTact/StandaloneArduino/blob/master/SingleTactDemo.ino (accessed Mar. 23, 2023).
[73] 'StandaloneArduino/SingleTactMultiSensorDemo.ino at master · SingleTact/StandaloneArduino'. https://github.com/SingleTact/StandaloneArduino/blob/master/SingleTactMultiSensorDemo.ino (accessed Mar. 23, 2023).
[74] 'MIDIView - Free MIDI Monitor tool for Win & Mac'. https://hautetechnique.com/midi/midiview/ (accessed Mar. 23, 2023).
[75] 'MIDI Scope - Apps on Google Play'. https://play.google.com/store/apps/details?id=com.mobileer.example.midiscope&hl=en_US (accessed Mar. 23, 2023).
[76] 'Toasts overview', *Android Developers*. https://developer.android.com/guide/topics/ui/notifiers/toasts (accessed Mar. 24, 2023).
[77] 'Linking External Controllers'. https://www.image-line.com/fl-studio-learning/fl-studio-online-manual/html/automation_linking.htm (accessed Mar. 27, 2023).
[78] M. Username, 'Answer to "How difficult is to get basic melody out of Theremin?"', *Music: Practice & Theory Stack Exchange*, Jan. 21, 2015. https://music.stackexchange.com/a/28966 (accessed Mar. 27, 2023).

[79] RBRODER, ‘Who here knows how to play the Theremin? Hard or easy to learn?’, *r/synthesizers*, Sep. 12, 2015. www.reddit.com/r/synthesizers/comments/3km2vh/who_here_knows_how_to_play_the_theremin_hard_or/ (accessed Mar. 27, 2023).

[80] ‘The 11 Hardest Instruments To Play, Learn (And Master)’, Jun. 12, 2022. https://producerhive.com/ask-the-hive/hardest-instruments-to-play-and-learn/ (accessed Mar. 27, 2023).

Appendix A – Statement Of Intent and Interim Report

A1: Statement Of Intent

A2: Interim Report

Appendix B –Code

B1: Microcontroller Code

B1.1: TMC_2.ino:

```
#include <Wire.h>  //For I2C/SMBus
#include <USB-MIDI.h> //Library for sending MIDI messages

USBMIDI_CREATE_DEFAULT_INSTANCE(); //required line to initialise MIDI communication

#define pinEcho 18 //Pin 18 on the Arduino Micro
#define pinTrig 19 //Pin 19 on the Arduino Micro, both used for the HC-SR04

#define range_cm 4 //range of each note
#define max_dist 8 * range_cm - 1  //max distance used = max note played, max_dist * 8 for white keys and max_dist * 13 for white + black keys
#define distance_offset 5 //offset for the initial reading of HC-SR04

#define buffer_size 50 //size of array buffer for the rolling average

#define sensor1_address 0x06  // Peripheral addresses used for SingleTact (SingleTact), default 0x04
#define sensor2_address 0x08
#define sensor3_address 0x0A

//the control change values that each SingleTact sensor will send
#define ST_left_CC 13
#define ST_middle_CC 12
#define ST_right_CC 7

#define enable_pitch_bend 0 //a future setting for only producing bended notes for a more Theremin like experience

//sensor measurements
int dist = 0;
short data1 = 0;
short data2 = 0;
short data3 = 0;

//checks and limits
int buffer[buffer_size] = { 0 }; //initialise an array of size buffer_size
int avg_dist = 0;
int weighted_avg_dist = 0;

```

```
//note data
int prev_note = 0;
int curr_note = 0;
int velocity = 0;

//for turning on or off the TMC based on input
char idle = 0;
bool on = true;

//customisations
int pitchBend_mode = enable_pitch_bend;
int pitchBend_val = 0;

//function initialisations
int* popBuffer(int data, int buffer[buffer_size]);
int calcAvg(int buffer[buffer_size]);

void setup() {
  setupUSR();
  MIDI.begin(1); //begin MIDI on channel 1
  Wire.begin();  // join i2c bus (address optional for master)
  Serial.begin(31250);  //serial for output, 31250 for correct MIDI output
  Serial.flush(); //waits for the transmission of outgoing serial data to complete
}

void loop() {

  measureAndCCsend();

  if (dist < max_dist) { //enter only if the measure distance is less than the max distance
    idle = 0;  //idle state set to false to signify it is working
    on = true;

    avg_dist = calcAvg(popBuffer(dist, buffer)); //calculate the average distance
    weighted_avg_dist = (1.0 * avg_dist) + (0.0 * dist); //calculate the weighted distance

    curr_note = noteOut_Cmaj(weighted_avg_dist); //set the current note to be the value of the noteOut function with the weighted average distance

    if (curr_note != prev_note && curr_note > 20 && pitchBend_mode == 0) {  //>20 is to prevent other random notes occuring when the variables are empty

```

```
      MIDI.sendNoteOff(prev_note, 127, 1); //turn off the previous note
      MIDI.sendNoteOn(curr_note, 127, 1); //send the current note

    } else if (pitchBend_mode == 1) { //code only executes if pitch bend mode is enabled but no functionality is provided to the user

      MIDI.sendNoteOff(prev_note, 127, 1); //do the same thing as before but
      curr_note = 48; //always set the current note to be number 48
      MIDI.sendNoteOn(curr_note, 127, 1);

      pitchBend_mode = 2; //set to 2 so that only pitch bend values get sent in the next iteration

    } else if (pitchBend_mode == 2) {

      pitchBend_val = map(weighted_avg_dist, 0, max_dist, MIDI_PITCHBEND_MIN, MIDI_PITCHBEND_MAX); ////use the map function to map the weighted distance between 0 to 16383

      MIDI.sendPitchBend(pitchBend_val, 1); //send the pitch bend value
    }
    prev_note = curr_note;  //gets stored before and after each check
  } else if (on) {          //if the distance is way too big it means no hand on top of USR
    idle++; //idle counter gets iterated
    if (idle > 100) { //if it reaches 100
      reset(); //reset the TMC
      pitchBend_mode = enable_pitch_bend; //reset the pitch bend value to default
    }
  }
}

void setupUSR() { //function to set up the connections to the HC-SR04
  pinMode(pinEcho, INPUT); //set pin 18 to be an input pin
  pinMode(pinTrig, OUTPUT); //set pin 19 to be an output pin
}

int getDistance() { //function to measure distance using HC-SR)4
  int _dist = 0;
  digitalWrite(pinTrig, LOW);  //give a short low pulse to have a cleaner high pulse
  delayMicroseconds(2);
  digitalWrite(pinTrig, HIGH);
  delayMicroseconds(10);       //give 10 us high pulse on the trigger pin
  digitalWrite(pinTrig, LOW);  //reset to low

  //speed of sound is 343 m/s so it is 0.0343 cm/us, divided by 2 to find distance,
  _dist = (pulseIn(pinEcho, HIGH) * 0.0343) / 2.0;  //function returns microseconds

  _dist -= distance_offset; //subtract the distance offset to achieve the lowest measurement of 0
  if (_dist < 0) { _dist = 0; } //if it goes below 0 then set it to 0


  return _dist;
}

short readDataFromSensor(short address) { //function to read data from SingleTact sensors
  int i2cPacketLength = 6;    //i2c packet length. Just need 6 bytes from each slave
  byte outgoingI2CBuffer[3];  //outgoing array buffer
  byte incomingI2CBuffer[6];  //incoming array buffer

  outgoingI2CBuffer[0] = 0x01;             //I2c read command
  outgoingI2CBuffer[1] = 128;              //Slave data offset
  outgoingI2CBuffer[2] = i2cPacketLength;  //require 6 bytes

  Wire.beginTransmission(address);             // transmit to device
  Wire.write(outgoingI2CBuffer, 3);            // send out command
  byte error = Wire.endTransmission();         // stop transmitting and check slave status
  if (error != 0) return -1;                   //if slave not exists or has error, return -1
  Wire.requestFrom(address, i2cPacketLength);  //require 6 bytes from slave

  byte incomeCount = 0;
  while (incomeCount < i2cPacketLength)  // slave may send less than requested
  {
    if (Wire.available()) {
      incomingI2CBuffer[incomeCount] = Wire.read();  // receive a byte as character
      incomeCount++;
    } else {
      delayMicroseconds(10);  //Wait 10us
    }
  }

  short rawData = (incomingI2CBuffer[4] << 8) + incomingI2CBuffer[5];  //get the raw data

  //used for debugging
  // Serial.println("Index 0:");
  // Serial.println(incomingI2CBuffer[0]);
  // Serial.println("Index 1:");
  // Serial.println(incomingI2CBuffer[1]);
  // Serial.println("Index 2:");
  // Serial.println(incomingI2CBuffer[2]);
  // Serial.println("Index 3:");
  // Serial.println(incomingI2CBuffer[3]);
  // Serial.println("Index 4:");
  // Serial.println(incomingI2CBuffer[4]);
  // Serial.println("Index 5:");
  // Serial.println(incomingI2CBuffer[5]);
  // Serial.println("\n");
  //Serial.println(rawData);

  return rawData;
}

void testSensorOutputs(int dist, short data1, short data2, short data3) { //function to give sensor outputs in Arduino IDE
  Serial.print("HC-SR04 Data:");
  Serial.print(dist); //output the corresponding variable with the correct label
  Serial.print("cm\n");

  Serial.print("I2C Sensor 1 Data:");
  Serial.print(data1);
  Serial.print("\n");

  Serial.print("I2C Sensor 2 Data:");
  Serial.print(data2);
  Serial.print("\n");

  Serial.print("I2C Sensor 3 Data:");
  Serial.print(data3);
  Serial.print("\n");
}

int noteOut(int dist) { //function that outputs the MIDI note value based on the distance measured
  if (dist <= range_cm * 0 && dist <= (1 * range_cm) - 1) { //range variable is used here to calculate the space each note will take
```

```
    return 48;  //C3
  } else if (dist <= range_cm * 1 && dist <= (2 * range_cm) - 1) {
    return 49;  //C#3
  } else if (dist <= range_cm * 2 && dist <= (3 * range_cm) - 1) {
    return 50;  //D3
  } else if (dist <= range_cm * 3 && dist <= (4 * range_cm) - 1) {
    return 51;  //D#3
  } else if (dist <= range_cm * 4 && dist <= (5 * range_cm) - 1) {
    return 52;  //E3
  } else if (dist <= range_cm * 5 && dist <= (6 * range_cm) - 1) {
    return 53;  //F3
  } else if (dist <= range_cm * 6 && dist <= (7 * range_cm) - 1) {
    return 54;  //F#3
  } else if (dist <= range_cm * 7 && dist <= (8 * range_cm) - 1) {
    return 55;  //G3
  } else if (dist <= range_cm * 8 && dist <= (9 * range_cm) - 1) {
    return 56;  //G#3
  } else if (dist <= range_cm * 9 && dist <= (10 * range_cm) - 1) {
    return 57;  //A3
  } else if (dist <= range_cm * 10 && dist <= (11 * range_cm) - 1) {
    return 58;  //A#3
  } else if (dist <= range_cm * 11 && dist <= (12 * range_cm) - 1) {
    return 59;  //B3
  } else if (dist <= range_cm * 12 && dist <= (13 * range_cm) - 1) {
    return 60;  //C4
  }
}

int noteOut_Cmaj(int dist) { //same as the noteOut function but here it is tuned to play C major
  if (dist <= range_cm * 0 && dist <= (1 * range_cm) - 1) {
    return 48;  //C3
  } else if (dist <= range_cm * 1 && dist <= (2 * range_cm) - 1) {
    return 50;  //D3
  } else if (dist <= range_cm * 2 && dist <= (3 * range_cm) - 1) {
    return 52;  //E3
  } else if (dist <= range_cm * 3 && dist <= (4 * range_cm) - 1) {
    return 53;  //F3
  } else if (dist <= range_cm * 4 && dist <= (5 * range_cm) - 1) {
    return 55;  //G3
  } else if (dist <= range_cm * 5 && dist <= (6 * range_cm) - 1) {
    return 57;  //A3
  } else if (dist <= range_cm * 6 && dist <= (7 * range_cm) - 1) {
    return 59;  //B3
  } else if (dist <= range_cm * 7 && dist < (8 * range_cm) - 1) {
    return 60;  //C4
  }
}

int* popBuffer(int data, int buffer[buffer_size]) { //function to populate the buffer array one value at a time
  for (int i = buffer_size - 1; i > 0; i--) {
    buffer[i] = buffer[i - 1]; //shift the value in index i, one to the right
  }
  buffer[0] = data; //assign the new data to the first value
  return buffer;
}

int calcAvg(int buffer[buffer_size]) { //calculate the average by taking the buffer array as an input
  int total = 0; //initialise the total to be 0
  for (int i = 0; i < buffer_size; i++) {
    total += buffer[i]; //add each value in the array to the total
  }
  return total / buffer_size; //divide by the buffer size to get the average
}

void reset() { //function to reset the TMC
  on = false;
  for (int i = 48; i <= 60; i++) {  //turn all notes off
    MIDI.sendNoteOff(i, 127, 1);
  }
}

void measureAndCCsend(){ //function that gets measurements from the sensors and sends the corresponing Control Change numbers
  dist = getDistance();
  data1 = map(readDataFromSensor(sensor1_address), 260, 1022, 0, 127);  //some calibration on the sensor data
  data2 = map(readDataFromSensor(sensor2_address), 256, 1022, 0, 127);
  data3 = map(readDataFromSensor(sensor3_address), 260, 1022, 0, 127);

  if (data3 > 4) { //only send data more than 4 to minimise noise
    MIDI.sendControlChange(7, data3, 1);
  } else {
    data3 = 0;
  }
  if (data2 > 4) {
    MIDI.sendControlChange(12, data2, 1);
  } else {
    data2 = 0;
  }
  if (data1 > 4) {
    MIDI.sendControlChange(13, data1, 1);
  } else {
    data1 = 0;
  }
}
```

B2: Android Application Code

B2.1 SelectionConfig.java

```java
package com.strath.tmc;

//class created to store the user configurations
public class SelectionConfig{

    private static boolean set;
    private static int sound_id;
    private static int effect_id1;
    private static int effect_id2;

    //class method to initialise the static class
    public static void init(boolean set_value, int sound_id_value,
int effect_id1_value, int effect_id2_value) {
        set = set_value;
        sound_id = sound_id_value;
        effect_id1 = effect_id1_value;
        effect_id2 = effect_id2_value;
    }

    //getters and setters for each variable
    public static boolean isSet() {
        return set;
    }

    public static void setSet(boolean set_value) {
        set = set_value;
    }

    public static int getSound_id() {
        return sound_id;
    }

    public static void setSound_id(int sound_id_value) {
        sound_id = sound_id_value;
    }

    public static int getEffect_id1() {
        return effect_id1;
    }

    public static void setEffect_id1(int effect_id1_value) {
        effect_id1 = effect_id1_value;
    }

    public static int getEffect_id2() {
        return effect_id2;
    }

    public static void setEffect_id2(int effect_id2_value) {
effect_id2 = effect_id2_value; }

}

```

B2.2 MainActivity.java

```java
package com.strath.tmc; //unique package name

import android.content.pm.PackageManager;
import androidx.appcompat.app.AppCompatActivity;

//imports for the Graphical User Interface
import android.os.Bundle;
import android.view.View;
import android.widget.CheckBox;
import android.widget.Spinner;
import android.widget.Toast;
import android.widget.ToggleButton;

//imports for MIDI functionality
import android.media.midi.MidiDevice.MidiConnection;
import android.media.midi.MidiDeviceInfo;
import android.media.midi.MidiManager;

//imports from the provided Android Example
import com.example.android.common.midi.MidiOutputPortConnectionSelector;
import com.example.android.common.midi.MidiPortConnector;
import com.example.android.common.midi.MidiTools;

public class MainActivity extends AppCompatActivity {
    //variable initialisations
    boolean set = false;
    boolean effects_selected = false, sound_selected = false, spinner_selected = false;

    private MidiOutputPortConnectionSelector mPortSelector;




    @Override
    protected void onCreate(Bundle savedInstanceState) { //function is called when the app is opened
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        SelectionConfig.init(false, 0, 0, 0); //initialise a SelectionConfig class

        //create variables for the spinner and the toggle button
        Spinner midi_spinner = (Spinner) findViewById(R.id.spinner_synth_sender);
        ToggleButton toggle = (ToggleButton) findViewById(R.id.tbSetReset); //initialise a toggle button
        //disable them
        toggle.setEnabled(false);
        midi_spinner.setEnabled(false);

        if (getPackageManager().hasSystemFeature(PackageManager.FEATURE_MIDI)) { //if statement to check if MIDI is supported
            setupMidi();
            //disable spinner
            midi_spinner.setEnabled(false);
            midi_spinner.setSelection(0);
            //disable toggle
            toggle.setAlpha(0.5f);
            toggle.setEnabled(false);
        } else{
            Toast.makeText(MainActivity.this, "MIDI not supported!", Toast.LENGTH_LONG).show(); //output an error if not supported

        }
    }

    private void setupMidi() {
        //variable initialisations required to setup the MIDI connection
        MidiManager midiManager;
        midiManager = (MidiManager) getSystemService(MIDI_SERVICE);
        MidiDeviceInfo synthInfo = MidiTools.findDevice(midiManager, "AndroidTest", "SynthExample");
        int portIndex = 0;
        mPortSelector = new MidiOutputPortConnectionSelector(midiManager, this, R.id.spinner_synth_sender, synthInfo, portIndex);
        mPortSelector.setConnectedListener(new MyPortsConnectedListener());
    }

    private void closeSynthResources() { //to close the port
        if (mPortSelector != null) {
            mPortSelector.close();
        }
    }

    private class MyPortsConnectedListener implements MidiPortConnector.OnPortsConnectedListener {
        @Override
        public void onPortsConnected(final MidiConnection connection) {
            runOnUiThread(() -> {
                if (connection == null) {
                    Toast.makeText(MainActivity.this, "Error: Device Not Connected", Toast.LENGTH_LONG).show();
                    mPortSelector.clearSelection();
                } else {
                    Toast.makeText(MainActivity.this, "Device Connected", Toast.LENGTH_LONG).show();
                }
            });
        }
    }

    @Override
    public void onDestroy() { //to stop everything
        closeSynthResources();
        super.onDestroy();
    }



    public void onCheckboxClicked(View view) { //when the checkbox is clicked this function executes

        //create variables for each checkbox and the toggle button
        CheckBox cb_sin = findViewById(R.id.cb_sin);
        CheckBox cb_saw = findViewById(R.id.cb_saw);

        CheckBox cb_pitch = findViewById(R.id.cb_pitch);
        CheckBox cb_duration = findViewById(R.id.cb_duration);
        CheckBox cb_bend_up = findViewById(R.id.cb_bend_up);
        CheckBox cb_bend_down = findViewById(R.id.cb_bend_down);

        ToggleButton toggle = (ToggleButton) findViewById(R.id.tbSetReset); //initialise a toggle button

        //code for setting sound id in config class and disabling the correct checkbox
        if(cb_sin.isChecked()){ //if it is checked, disable the other box
            cb_saw.setEnabled(false);
            SelectionConfig.setSound_id(1);
            sound_selected = true;
        } else if(cb_saw.isChecked()){
            cb_sin.setEnabled(false);
            SelectionConfig.setSound_id(2);
            sound_selected = true;
        } else{
            cb_sin.setEnabled(true);
            cb_saw.setEnabled(true);
            sound_selected = false;
        }

        //to set effect checkbox id in config class
        //this block is for the pitch effect with effect ID of 1
        if(cb_pitch.isChecked()){ //if the checkbox is checked
            if(SelectionConfig.getEffect_id1() == 0){ //only set values if they are 0
                SelectionConfig.setEffect_id1(1); //set the effect id
                //only set the value of the second effect is the current value of effect id 2 is 0 and effect id 1 is not the selected checkbox
            } else if(SelectionConfig.getEffect_id2() == 0 && SelectionConfig.getEffect_id1() != 1 ){
                SelectionConfig.setEffect_id2(1);
            }
        } else if (SelectionConfig.getEffect_id1() == 1){ //checks if when checkbox is unchecked, the effect value set is the one selected
            SelectionConfig.setEffect_id1(0); //if it is then reset the value back to 0
        } else if (SelectionConfig.getEffect_id2() == 1){
            SelectionConfig.setEffect_id2(0);
        }

        //this block is for the duration effect with effect ID of 2
        if(cb_duration.isChecked()) { //same as previous happens for each checkbox
            if (SelectionConfig.getEffect_id1() == 0) {
                SelectionConfig.setEffect_id1(2);
            } else if(SelectionConfig.getEffect_id2() == 0 && SelectionConfig.getEffect_id1() != 2 ){
                SelectionConfig.setEffect_id2(2);
            }
        } else if (SelectionConfig.getEffect_id1() == 2){
            SelectionConfig.setEffect_id1(0);
        } else if (SelectionConfig.getEffect_id2() == 2){
            SelectionConfig.setEffect_id2(0);
        }

        //this block is for the pitch bend up effect with effect ID of 3
        if (cb_bend_up.isChecked()) {
            if (SelectionConfig.getEffect_id1() == 0) {
                SelectionConfig.setEffect_id1(3);
            } else if(SelectionConfig.getEffect_id2() == 0 && SelectionConfig.getEffect_id1() != 3 ){
                SelectionConfig.setEffect_id2(3);
            }
        } else if (SelectionConfig.getEffect_id1() == 3){
            SelectionConfig.setEffect_id1(0);
        } else if (SelectionConfig.getEffect_id2() == 3){
            SelectionConfig.setEffect_id2(0);
        }

        //this block is for the pitch bend down effect with effect ID of 4
        if (cb_bend_down.isChecked()) {
            if (SelectionConfig.getEffect_id1() == 0) {
                SelectionConfig.setEffect_id1(4);
            } else if(SelectionConfig.getEffect_id2() == 0 && SelectionConfig.getEffect_id1() != 4 ){
                SelectionConfig.setEffect_id2(4);
            }
        } else if (SelectionConfig.getEffect_id1() == 4){
            SelectionConfig.setEffect_id1(0);
        } else if (SelectionConfig.getEffect_id2() == 4){
            SelectionConfig.setEffect_id2(0);
        }

        //code to disable the correct checkboxes after selection
        if(cb_pitch.isChecked() && cb_duration.isChecked()){
            cb_bend_up.setEnabled(false);
            cb_bend_down.setEnabled(false);
            effects_selected = true;
        } else if(cb_pitch.isChecked() && cb_bend_up.isChecked()){
            cb_duration.setEnabled(false);
            cb_bend_down.setEnabled(false);
            effects_selected = true;
        } else if(cb_pitch.isChecked() && cb_bend_down.isChecked()){
            cb_duration.setEnabled(false);
            cb_bend_up.setEnabled(false);
            effects_selected = true;
        } else if(cb_duration.isChecked() && cb_bend_up.isChecked()){
            cb_pitch.setEnabled(false);
            cb_bend_down.setEnabled(false);
            effects_selected = true;
        }else if(cb_duration.isChecked() && cb_bend_down.isChecked()){
            cb_pitch.setEnabled(false);
            cb_bend_up.setEnabled(false);
            effects_selected = true;
```

```java
        }else if(cb_bend_up.isChecked() && cb_bend_down.isChecked()){
            cb_pitch.setEnabled(false);
            cb_duration.setEnabled(false);
            effects_selected = true;
        } else{
            cb_pitch.setEnabled(true);
            cb_duration.setEnabled(true);
            cb_bend_up.setEnabled(true);
            cb_bend_down.setEnabled(true);
            effects_selected = false;
        }

        //check if effects and sounds are selected
        if((effects_selected) && (sound_selected)){ //if true, enable the spinner
            toggle.setAlpha(1f);
            toggle.setEnabled(true);
        } else { //if false, disable it and reset selection
            toggle.setAlpha(0.5f);
            toggle.setEnabled(false);
        }

        //for debugging
        //Toast.makeText(MainActivity.this, String.valueOf(SelectionConfig.getEffect_id1()), Toast.LENGTH_SHORT).show();
        //Toast.makeText(MainActivity.this, String.valueOf(SelectionConfig.getEffect_id2()), Toast.LENGTH_SHORT).show();
    }


    public void onButtonClicked(View view) {
        //create variables for each checkbox, the toggle button, and the spinner
        CheckBox cb_sin = findViewById(R.id.cb_sin);
        CheckBox cb_saw = findViewById(R.id.cb_saw);

        CheckBox cb_pitch = findViewById(R.id.cb_pitch);
        CheckBox cb_duration = findViewById(R.id.cb_duration);
        CheckBox cb_bend_up = findViewById(R.id.cb_bend_up);
        CheckBox cb_bend_down = findViewById(R.id.cb_bend_down);

        ToggleButton toggle = (ToggleButton) findViewById(R.id.tbSetReset); //initialise a toggle button

        Spinner midi_spinner = (Spinner) findViewById(R.id.spinner_synth_sender);
```

```

      if(SelectionConfig.isSet()){ //if the button is pressed when "Reset" is shown
        //uncheck all check boxes
        cb_sin.setChecked(false);
        cb_saw.setChecked(false);
        cb_pitch.setChecked(false);
        cb_duration.setChecked(false);
        cb_bend_up.setChecked(false);
        cb_bend_down.setChecked(false);

        //enable all checkboxes
        cb_sin.setEnabled(true);
        cb_saw.setEnabled(true);
        cb_pitch.setEnabled(true);
        cb_duration.setEnabled(true);
        cb_bend_up.setEnabled(true);
        cb_bend_down.setEnabled(true);

        //disable the toggle button just like in startup
        toggle.setAlpha(0.5f);
        toggle.setEnabled(false);

        //reset selector variables
        effects_selected = false;
        sound_selected = false;
        spinner_selected = false;

        //disable spinner
        midi_spinner.setEnabled(false);
        midi_spinner.setSelection(0);

        //reset all variables
        SelectionConfig.setSound_id(0);
        SelectionConfig.setEffect_id1(0);
        SelectionConfig.setEffect_id2(0);

        } else{ //if it is pressed when "Set" is shown
        //disable all checkboxes
        cb_sin.setEnabled(false);
        cb_saw.setEnabled(false);
        cb_pitch.setEnabled(false);
        cb_duration.setEnabled(false);
        cb_bend_up.setEnabled(false);
        cb_bend_down.setEnabled(false);

        //enable spinner
        midi_spinner.setEnabled(true);

```

```java
      //error checking in case the values get mixed up and the same effect is selected twice
      if(SelectionConfig.getEffect_id1() == SelectionConfig.getEffect_id2() && effects_selected) {
        //reset both effect ids to 0
        SelectionConfig.setEffect_id1(0);
        SelectionConfig.setEffect_id2(0);

        //uncheck all effects check boxes
        cb_pitch.setChecked(false);
        cb_duration.setChecked(false);
        cb_bend_up.setChecked(false);
        cb_bend_down.setChecked(false);

        //output an error as a Toast
        Toast.makeText(MainActivity.this, "ERROR: The same effects were detected. Please retry.", Toast.LENGTH_SHORT).show();
        effects_selected = false;

        //disable spinner
        midi_spinner.setEnabled(false);
        midi_spinner.setSelection(0);
      }

    }

    //pass the value of set to the class
    SelectionConfig.setSet(toggle.isChecked());

    //for debugging purposes
    //Toast.makeText(MainActivity.this, String.valueOf(SelectionConfig.getEffect_id1()), Toast.LENGTH_SHORT).show();
    //Toast.makeText(MainActivity.this, String.valueOf(SelectionConfig.getEffect_id2()), Toast.LENGTH_SHORT).show();

  }
}
```

B2.3 EnevelopeASDR.java

```java
/*
 * Copyright (C) 2015 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.android.common.midi.synth;

/**
 * Very simple Attack, Decay, Sustain, Release envelope with linear ramps.
 *
 * Times are in seconds.
 */
public class EnvelopeADSR extends SynthUnit {
    private static final int IDLE = 0;
    private static final int ATTACK = 1;
    private static final int DECAY = 2;
    private static final int SUSTAIN = 3;
    private static final int RELEASE = 4;
    private static final int FINISHED = 5;
    private static final float MIN_TIME = 0.001f;

    private float mAttackRate;
    private float mRreleaseRate;
    private static float mSustainLevel; //had to change this to static in order for the static function to work
    private float mDecayRate;
    private float mCurrent;
    private int mSstate = IDLE;

    public EnvelopeADSR() { //try to change this with the CCs
        setAttackTime(0.003f); //used to be 0.003f
        setDecayTime(0.08f);
        setSustainLevel(0.3f); //used to be 0.3f
        setReleaseTime(1.0f);
    }

    public void setAttackTime(float time) {
        if (time < MIN_TIME)
            time = MIN_TIME;
        mAttackRate = 1.0f / (SynthEngine.FRAME_RATE * time);
    }

    public void setDecayTime(float time) {
        if (time < MIN_TIME)
            time = MIN_TIME;
        mDecayRate = 1.0f / (SynthEngine.FRAME_RATE * time);
    }
    //-------------------ADDITIONS----------------------------------------
    //changed the function to static in order to access it from the SynthEngine class
    public static void setSustainLevel(float level) {
        if (level < 0.0f)
            level = 0.0f;
        mSustainLevel = level;
    }
    //----------------------------------------------
    public void setReleaseTime(float time) {
        if (time < MIN_TIME)
            time = MIN_TIME;
        mRreleaseRate = 1.0f / (SynthEngine.FRAME_RATE * time);
    }

    public void on() {
        mSstate = ATTACK;
    }

    public void off() {
        mSstate = RELEASE;
    }

    @Override
    public float render() {
        switch (mSstate) {
        case ATTACK:
            mCurrent += mAttackRate;
            if (mCurrent > 1.0f) {
                mCurrent = 1.0f;
                mSstate = DECAY;
            }
            break;
        case DECAY:
            mCurrent -= mDecayRate;
            if (mCurrent < mSustainLevel) {
                mCurrent = mSustainLevel;
                mSstate = SUSTAIN;
            }
            break;
        case RELEASE:
            mCurrent -= mRreleaseRate;
            if (mCurrent < 0.0f) {
                mCurrent = 0.0f;
                mSstate = FINISHED;
            }
            break;
        }
        return mCurrent;
    }

    public boolean isDone() {
        return mSstate == FINISHED;
    }

}

```

B2.4: SineVoice.java

```java
/*
 * Copyright (C) 2015 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.android.common.midi.synth;

/**
 * Replace sawtooth with a sine wave.
 */
public class SineVoice extends SawVoice {
    @Override
    protected SineOscillator createOscillator() {
        return new SineOscillator();
    }

    //---------ADDITIONS---------------------------------------------------------------------
    //created a Sine voice the same way the Saw voice was created but with the provided
    //Sine Oscillator class
    private SineOscillator mOscillator;
    private EnvelopeADSR mEnvelope;

    public SineVoice() {
        mOscillator = createOscillator();
        mEnvelope = new EnvelopeADSR();
    }

    @Override
    public void noteOn(int noteIndex, int velocity) {
        super.noteOn(noteIndex, velocity);
        mOscillator.setPitch(noteIndex);
        mOscillator.setAmplitude(getAmplitude());
        mEnvelope.on();
    }

    @Override
    public void noteOff() {
        super.noteOff();
        mEnvelope.off();
    }

    @Override
    public void setFrequencyScaler(float scaler) {
        mOscillator.setFrequencyScaler(scaler);
    }

    @Override
    public float render() {
        float output = mOscillator.render() * mEnvelope.render();
        return output;
    }

    @Override
    public boolean isDone() {
        return mEnvelope.isDone();
    }

//----------------------------------------------------------------------------------------------
}

```

B2.5: SineOscillator.java

```java
/*
 * Copyright (C) 2015 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.android.common.midi.synth;

/**
 * Sinewave oscillator.
 */
public class SineOscillator extends SawOscillator {
    // Factorial constants.
    private static final float IF3 = 1.0f / (2 * 3);
    private static final float IF5 = IF3 / (4 * 5);
    private static final float IF7 = IF5 / (6 * 7);
    private static final float IF9 = IF7 / (8 * 9);
    private static final float IF11 = IF9 / (10 * 11);

    /**
     * Calculate sine using Taylor expansion. Do not use values outside the range.
     *
     * @param currentPhase in the range of -1.0 to +1.0 for one cycle
     */
    public static float fastSin(float currentPhase) {

        /* Wrap phase back into region where results are more accurate. */
        float yp = (currentPhase > 0.5f) ? 1.0f - currentPhase
                : ((currentPhase < (-0.5f)) ? (-1.0f) - currentPhase : currentPhase);

        float x = (float) (yp * Math.PI);
        float x2 = (x * x);
        /* Taylor expansion out to x**11/11! factored into multiply-adds */
        return x * (x2 * (x2 * (x2 * (x2 * ((x2 * (-IF11)) + IF9) - IF7) + IF5) - IF3) + 1);
    }

    @Override
    public float render() {
        // Convert raw sawtooth to sine.
        float phase = incrementWrapPhase();
        return fastSin(phase) * getAmplitude();
    }

}

```

B2.6: SawVoice.java

```java
/*
 * Copyright (C) 2015 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.android.common.midi.synth;

/**
 * Sawtooth oscillator with an ADSR.
 */
public class SawVoice extends SynthVoice {
    private SawOscillator mOscillator;
    private EnvelopeADSR mEnvelope;

    public SawVoice() {
        mOscillator = createOscillator();
        mEnvelope = new EnvelopeADSR();
    }

    protected SawOscillator createOscillator() {
        return new SawOscillator();
    }

    @Override
    public void noteOn(int noteIndex, int velocity) {
        super.noteOn(noteIndex, velocity);
        mOscillator.setPitch(noteIndex);
        mOscillator.setAmplitude(getAmplitude());
        mEnvelope.on();
    }

    @Override
    public void noteOff() {
        super.noteOff();
        mEnvelope.off();
    }

    @Override
    public void setFrequencyScaler(float scaler) {
        mOscillator.setFrequencyScaler(scaler);
    }

    @Override
    public float render() {
        float output = mOscillator.render() * mEnvelope.render();
        return output;
    }

    @Override
    public boolean isDone() {
        return mEnvelope.isDone();
    }

}
```

B2.7: SawOscillator.java

```java
/*
 * Copyright (C) 2015 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.android.common.midi.synth;

public class SawOscillator extends SynthUnit {
    private float mPhase = 0.0f;
    private float mPhaseIncrement = 0.01f;
    private float mFrequency = 0.0f;
    private float mFrequencyScaler = 1.0f;
    private float mAmplitude = 1.0f;

    public void setPitch(float pitch) {
        float freq = (float) pitchToFrequency(pitch);
        setFrequency(freq);
    }

    public void setFrequency(float frequency) {
        mFrequency = frequency;
        updatePhaseIncrement();
    }

    private void updatePhaseIncrement() {
        mPhaseIncrement = 2.0f * mFrequency * mFrequencyScaler / 48000.0f;
    }

    public void setAmplitude(float amplitude) {
        mAmplitude = amplitude;
    }

    public float getAmplitude() {
        return mAmplitude;
    }

    public float getFrequencyScaler() {
        return mFrequencyScaler;
    }

    public void setFrequencyScaler(float frequencyScaler) {
        mFrequencyScaler = frequencyScaler;
        updatePhaseIncrement();
    }

    float incrementWrapPhase() {
        mPhase += mPhaseIncrement;
        while (mPhase > 1.0) {
            mPhase -= 2.0;
        }
        while (mPhase < -1.0) {
            mPhase += 2.0;
        }
        return mPhase;
    }

    @Override
    public float render() {
        return incrementWrapPhase() * mAmplitude;
    }

}
```

B2.8: SynthEngine.java

```java
/*
 * Copyright (C) 2015 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.example.android.common.midi.synth;

import android.media.midi.MidiReceiver;
import android.util.Log;

import com.example.android.common.midi.MidiConstants;
import com.example.android.common.midi.MidiEventScheduler;
import com.example.android.common.midi.MidiEventScheduler.MidiEvent;
import com.example.android.common.midi.MidiFramer;

import com.strath.tmc.SelectionConfig;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.Iterator;

/**
 * Very simple polyphonic, single channel synthesizer. It runs a background
 * thread that processes MIDI events and synthesizes audio dsfwserfawerfwaerwqae.
 */
public class SynthEngine extends MidiReceiver {

    private static final String TAG = "SynthEngine";

    public static final int FRAME_RATE = 48000;
    private static final int FRAMES_PER_BUFFER = 240;
    private static final int SAMPLES_PER_FRAME = 2;

    private boolean go;
    private Thread mThread;
    private float[] mBuffer = new float[FRAMES_PER_BUFFER * SAMPLES_PER_FRAME];
    private float mFrequencyScaler = 1.0f;
    private int mProgram;

    private ArrayList<SynthVoice> mFreeVoices = new ArrayList<SynthVoice>();
    private Hashtable<Integer, SynthVoice>
            mVoices = new Hashtable<Integer, SynthVoice>();
    private MidiEventScheduler mEventScheduler;
    private MidiFramer mFramer;
    private MidiReceiver mReceiver = new MyReceiver();
    private SimpleAudioOutput mAudioOutput;

    //-------------------ADDITIONS-----------------------------------------
    //only variable additions to this java class
    public static int velocity_cc = 127; //set a constant value for velocity
    public static float changeSustain = 0.3f; //variable to use to adjust sustain and pass it onto the function
    private float mBendRange = 12.0f; // in semitones, used to be 2.0f but changed to allow for a wider bend range

    //---------------------------------------------------------


    public SynthEngine() {
        this(new SimpleAudioOutput());
    }

    public SynthEngine(SimpleAudioOutput audioOutput) {
        mReceiver = new MyReceiver();
        mFramer = new MidiFramer(mReceiver);
        mAudioOutput = audioOutput;
    }

    @Override
    public void onSend(byte[] data, int offset, int count, long timestamp)
            throws IOException {
        if (mEventScheduler != null) {
            if (!MidiConstants.isAllActiveSensing(data, offset, count)) {
                mEventScheduler.getReceiver().send(data, offset, count,
                        timestamp);
            }
        }
    }

    private class MyReceiver extends MidiReceiver {
        @Override
        public void onSend(byte[] data, int offset, int count, long timestamp)
                throws IOException {
            byte command = (byte) (data[0] & MidiConstants.STATUS_COMMAND_MASK);
            int channel = (byte) (data[0] & MidiConstants.STATUS_CHANNEL_MASK);
            switch (command) {
            case MidiConstants.STATUS_NOTE_OFF:
                noteOff(channel, data[1], data[2]);
                break;
            case MidiConstants.STATUS_NOTE_ON:
                noteOn(channel, data[1], velocity_cc);
                break;
            case MidiConstants.STATUS_PITCH_BEND:
                int bend = (data[2] << 7) + data[1];
                pitchBend(channel, bend); //change the bend variable to 12f to have a full octave
                break;
            case MidiConstants.STATUS_PROGRAM_CHANGE:
                mProgram = data[1];
                mFreeVoices.clear();
                break;
                //-------------------ADDITIONS-----------------------------------------
            case MidiConstants.STATUS_CONTROL_CHANGE: //used to check when a Control Change message is sent
                controlChange(channel, data[1], data[2]);
                break;
                //---------------------------------------------------------
            default:
                logMidiMessage(data, offset, count);
                break;
            }
        }
    }

    class MyRunnable implements Runnable {
        @Override
        public void run() {
            try {
                mAudioOutput.start(FRAME_RATE);
                onLoopStarted();
                while (go) {
                    processMidiEvents();
                    generateBuffer();
                    mAudioOutput.write(mBuffer, 0, mBuffer.length);
                    onBufferCompleted(FRAMES_PER_BUFFER);
                }
            } catch (Exception e) {
                Log.e(TAG, "SynthEngine background thread exception.", e);
            } finally {
                onLoopEnded();
                mAudioOutput.stop();
            }
        }
    }

    /**
     * This is called form the synthesis thread before it starts looping.
     */
    public void onLoopStarted() {
    }

    /**
     * This is called once at the end of each synthesis loop.
     *
     * @param framesPerBuffer
     */
    public void onBufferCompleted(int framesPerBuffer) {
    }

    /**
     * This is called form the synthesis thread when it stop looping.
     */
    public void onLoopEnded() {
    }

    /**
     * Assume message has been aligned to the start of a MIDI message.
     *
     * @param data
```

```java
     * @param offset
     * @param count
     */
    public void logMidiMessage(byte[] data, int offset, int count) {
        String text = "Received: ";
        for (int i = 0; i < count; i++) {
            text += String.format("0x%02X, ", data[offset + i]);
        }
        Log.i(TAG, text);
    }

    /**
     * @throws IOException
     *
     */
    private void processMidiEvents() throws IOException {
        long now = System.nanoTime(); // TODO use audio presentation time
        MidiEvent event = (MidiEvent) mEventScheduler.getNextEvent(now);
        while (event != null) {
            mFramer.send(event.data, 0, event.count, event.getTimestamp());
            mEventScheduler.addEventToPool(event);
            event = (MidiEvent) mEventScheduler.getNextEvent(now);
        }
    }

    /**
     *
     */
    private void generateBuffer() {
        for (int i = 0; i < mBuffer.length; i++) {
            mBuffer[i] = 0.0f;
        }
        Iterator<SynthVoice> iterator = mVoices.values().iterator();
        while (iterator.hasNext()) {
            SynthVoice voice = iterator.next();
            if (voice.isDone()) {
                iterator.remove();
                mFreeVoices.add(voice);
            } else {
                voice.mix(mBuffer, SAMPLES_PER_FRAME, 0.25f);
            }
        }
    }

    public void noteOff(int channel, int noteIndex, int velocity) {
        SynthVoice voice = mVoices.get(noteIndex);
        if (voice != null) {
            voice.noteOff();
        }
    }

    public void allNotesOff() {
        Iterator<SynthVoice> iterator = mVoices.values().iterator();
        while (iterator.hasNext()) {
            SynthVoice voice = iterator.next();
            voice.noteOff();
        }
    }

    /**
     * Create a SynthVoice.
     */
    public SynthVoice createVoice(int program) {
        //-------------------ADDITIONS-----------------------------------------
        // For every odd program number use a sine wave.
        program = SelectionConfig.getSound_id();
        //---------------------------------------------------------------------
        if (program == 1) {
            return new SineVoice();
        } else {
            return new SawVoice();
        }
    }

    /**
     *
     * @param channel
     * @param noteIndex
     * @param velocity
     */
    public void noteOn(int channel, int noteIndex, int velocity) {
        if (velocity == 0) {
            noteOff(channel, noteIndex, velocity);
        } else {
            mVoices.remove(noteIndex);
            SynthVoice voice;
            if (mFreeVoices.size() > 0) {
                voice = mFreeVoices.remove(mFreeVoices.size() - 1);
            } else {
                voice = createVoice(mProgram);
            }
            voice.setFrequencyScaler(mFrequencyScaler);
            voice.noteOn(noteIndex, velocity);
            mVoices.put(noteIndex, voice);

        }
    }
    //-------------------ADDITIONS---------------------------------------------------------------------------
    /** brief Send a Control Change message
    @param channel The channel on which the message will be sent (1 to 16).
    @param controlNum The control change number
    @param controlValue The value of the channel
    */
    public void controlChange(int channel, int controlNum, int controlValue ){ //function for handling Control Change messages
        switch(controlNum){
            case 7: //hardcoded for when control change 7 is used, it changes the velocity
                velocity_cc = 127 - controlValue;
                break;
            case 13: //for control change 13
                if (SelectionConfig.getEffect_id1() == 1) { //if effect id is one
                    if (controlValue > 10) { //and if its CC value is more than 10
                        mFrequencyScaler = 1.0f + (float)(controlValue/16); //add it to the variable to scale the frequency
                    } else{
                        mFrequencyScaler = 1.0f; //else set it to default
                    }
                } else if (SelectionConfig.getEffect_id1() == 2){ //if effect id is 2
                    if(controlValue > 10) { //and control value is over 20
                        changeSustain = (float)(controlValue / 256); //change the sustain
                        EnvelopeADSR.setSustainLevel(changeSustain); //pass it to the function
                    } else {
                        EnvelopeADSR.setSustainLevel(0.3f); //else set it to the default value
                    }
                } else if (SelectionConfig.getEffect_id1() == 3){ //if effect 3 is selected
```

```java
                        pitchBend(0, (int) map(controlValue, 3, 127, 8192, 16384)); //map the value to the correct range, and pass it as a pitch bend value
                    } else if (SelectionConfig.getEffect_id1() == 4){ //same happens here
                        pitchBend(0, (int) map(controlValue, 3, 127, 8192, 0));
                    }
                    break;
                case 12: //for control change 13 the same thing happens when control change 12 is detected
                    if (SelectionConfig.getEffect_id2() == 1) {
                        if (controlValue > 10) {
                            mFrequencyScaler = 1.0f + (float)(controlValue/16);
                        } else{
                            mFrequencyScaler = 1.0f;
                        }
                    } else if (SelectionConfig.getEffect_id2() == 2){
                        if(controlValue > 10) {
                            changeSustain = (float) (controlValue / 256);
                            EnvelopeADSR.setSustainLevel(changeSustain);
                        } else {
                            EnvelopeADSR.setSustainLevel(0.3f);
                        }
                    } else if (SelectionConfig.getEffect_id2() == 3){
                            pitchBend(0, (int) map(controlValue, 3, 127, 8192, 16384));
                    } else if (SelectionConfig.getEffect_id2() == 4){
                            pitchBend(0, (int) map(controlValue, 3, 127, 8192, 0));
                    }
                    break;
                default:
                    break;
            }
    }
    //taken from Arduino documentation
    public long map(int x, int in_min, int in_max, int out_min, int out_max) {
        return (long) (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
    }
//---------------------------------------------------------------------------------------------------------------------------------
    public void pitchBend(int channel, int bend) {
        double semitones = (mBendRange * (bend - 0x2000)) / 0x2000;
        mFrequencyScaler = (float) Math.pow(2.0, semitones / 12.0);
        Iterator<SynthVoice> iterator = 
mVoices.values().iterator();
        while (iterator.hasNext()) {
            SynthVoice voice = iterator.next();
            voice.setFrequencyScaler(mFrequencyScaler);
        }
    }

    /**
     * Start the synthesizer.
     */
    public void start() {
        stop();
        go = true;
        mThread = new Thread(new MyRunnable());
        mEventScheduler = new MidiEventScheduler();
        mThread.start();
    }

    /**
     * Stop the synthesizer.
     */
    public void stop() {
        go = false;
        if (mThread != null) {
            try {
                mThread.interrupt();
                mThread.join(500);
            } catch (InterruptedException e) {
                // OK, just stopping safely.
            }
            mThread = null;
            mEventScheduler = null;
        }
    }
}

```

B2.9: AndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-feature
        android:name="android.software.midi"
        android:required="true"/>


    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/tmc"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/Theme.TMC"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <!-- -ADDITIONS-
        Added the following ten lines in order for the application
to function
        Taken from the Android Example
        -->
        <service
            android:name=".MidiSynthDeviceService"
            android:permission="android.permission.BIND_MIDI_DEVICE_
SERVICE"
            android:exported="true">
            <intent-filter>
                <action
android:name="android.media.midi.MidiDeviceService"/>
            </intent-filter>
            <meta-data
                android:name="android.media.midi.MidiDeviceService"
                android:resource="@xml/synth_device_info"/>
        </service>

    </application>

```

```
</manifest>
```