University of Sheffield

A Modular Synthesiser Using the Raspberry Pi

Benjamin Simon Clegg

Supervisor: Professor Guy Brown

A report submitted in fulfilment of the requirements
for the degree of BSc in Computer Science

in the

Department of Computer Science

May 3, 2017

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: *Benjamin Simon Clegg*

Signature:

Date: May 3, 2017

Abstract

The goal of this project is to create a synthesiser which combines the creative freedom of a modular system, and the portability of a performance synthesiser, whilst being able to be produced at a low cost, by performing signal processing using the Raspberry Pi, an inexpensive playing card sized computer.

The challenges and processes involved in building such a synthesiser are outlined in this report, with an examination into existing synthesis techniques and resources which can be employed to successfully implement a modular synthesiser in software.

Acknowledgements

I would like to sincerely thank my supervisor, Professor Guy Brown, for his excellent advice and supervision.

Thanks to Abi for ensuring that I ate more than instant ramen, and maintained focus towards the end of the project.

I also wish to thank my parents for trusting that I wouldn't burn their house down with a soldering iron.

Finally, I'd like to thank all of my friends and family for their continuous moral support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modular synthesisers are a series of individual audio processing units, called modules, which take electrical signals as inputs and use them as outputs. These electrical currents have analogue voltages which can be converted into an audio output.

Each module is capable of generating or modifying an audio signal. For example, some modules are oscillators, which generate simple sound waves, other modules could be filters, which filter out particular sound frequencies.

Today, electronic synthesisers are widely used in music production and live performances. However, typical performance synthesisers are often limited in the sounds which they can produce, due to their unchangeable internal circuitry.

The nature of modular synthesisers allows for their audio processing units to be chained together in interesting manners - the exact same modules can be used in different combinations to create entirely different sounds. This provides more freedom over sound design compared to conventional synthesisers.

However, modular synthesisers are much larger than conventional synthesisers, and are therefore less suitable for live performances.

A synthesiser which offers a compromise between a performance synthesiser's portability and the freedom of a modular synthesiser would be particularly beneficial. Semi-modular synthesisers, such as the Moog Mother-32 (figure 1.1) offer this functionality. [5]

Due to its small size and configurable input, the Raspberry Pi computer could be used to implement such a synthesiser.

1.1 Aims and Objectives

The key aim of this project was to create a portable synthesiser using a Raspberry Pi, featuring a design inspired by modular synthesisers. The Raspberry Pi performs audio processing in software, where each "module" is implemented as a subroutine in a program.

This project also investigates the processes and challenges involved in implementing a synthesiser in such a manner.

Figure 1.1: A Moog Mother-32 semi-modular synthesiser [1]

The project also featured two additional objectives; for the resulting synthesiser to be appropriate for use as a resource for music and computer education, and to provide any developed software freely to other individuals (such as hobbyists) who wish to implement a similar system.

1.2 Overview of the Report

This report first outlines the research undertaken in preparation of the project, including synthesis techniques, musical interfaces, hardware limitations, and relevant software.

The requirements of the project are then outlined, with a system plan drafted to outline the basic design of the system. The stakeholders of the project, alongside potential ethical and legal concerns are considered, and a plan to evaluate the system is described.

Next, various design decisions are explained, and the processes involved in designing various manufacturing materials are highlighted. The software architecture is also described, and specific schematics are shown.

The processes and challenges involved in implementing the system are then described in detail.

Finally, the synthesiser is evaluated by performing an analysis of the performed study, potential further work is outlined, and the achievements and limitations of the project are discussed.

Chapter 2

Literature Survey

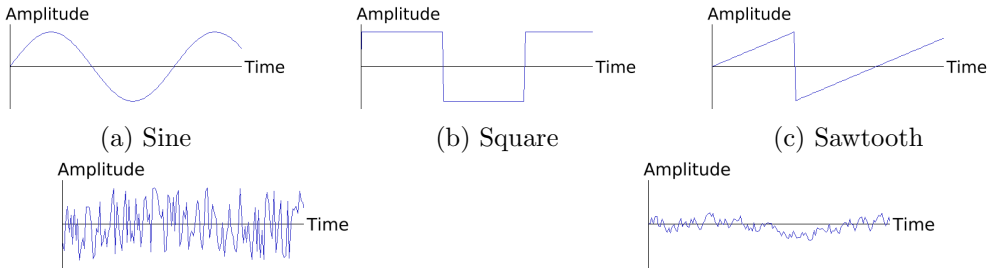
2.1 Synthesisers

Synthesisers are electronic instruments which produce audio signals, which can ultimately be converted to physical soundwaves using an amplifier and a speaker.

Inputs are often taken in the form of a keyboard, either integrated into the synthesiser itself (such as the *Yamaha DX7*), or over an interface (such as MIDI). These inputs change the frequency of the synthesised sounds, producing a note corresponding to the key pressed. [2, p. 4]

2.1.1 Waveforms

Synthesisers often use one or more wave functions as the foundation of a sound, which is later processed. The types of waveforms used can be selected by an operator, often including sawtooth, sine, and square waves. Pink or white noise are also common functions. [6] See figure 2.1 for examples.



(a) Sine (b) Square (c) Sawtooth

(d) White noise (e) Pink noise

Figure 2.1: Various waveforms, generated by *Audacity*

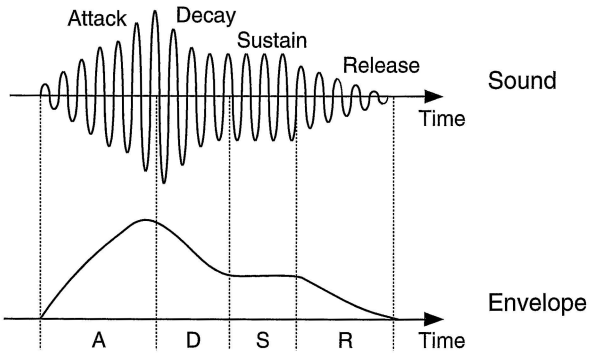


Figure 2.2: An ADSR Envelope [2, p. 39]

2.1.2 Envelopes (ADSR)

The amplitude of a synthesised sound is often controlled using an ADSR envelope (attack, decay, sustain, release).

As shown in figure 2.2, *attack* is the initial increase in amplitude to a peak. *Decay* is the decrease from this peak to an "equilibrium". *Sustain* is the time this equilibrium state is held for. *Release* is the reduction from this amplitude to zero.

The time and target amplitude of each stage can often be adjusted as a parameter. These envelopes can also control other properties, such as pitch. [2, pp. 38–39, 88–98]

2.2 Synthesis Approaches

Multiple approaches can be taken when designing a synthesiser, and various paradigms can be combined together to create a sound.

2.2.1 Subtractive

Subtractive synthesis involves taking a "broadband signal" (an audio signal containing a wide range of sound frequencies), and filtering it at particular frequencies to achieve a desired sound. The parameters of these filters can be varied with time to directly change the resulting signal. [7, pp. 263–264] [8]

2.2.2 Additive

Additive synthesis is based on upon Fourier's theorem, that "any periodic waveform that meets certain mathematical conditions can be expressed as the sum of a number ... of harmonically related sinusoids, each with a particular amplitude and phase." As such, by combining the outputs of multiple simple sine wave oscillators, of different amplitudes and phases, a complex sound can be synthesised. [7, pp. 208–209] [9]

2.2.3 Frequency Modulation (FM)

FM synthesis relies on using one oscillator to modulate the frequency of another. As this process can combine multiple soundwaves (which can also be formed by utilising other approaches), very diverse sounds can be produced. [10]

2.2.4 Wavetable

Wavetable synthesis uses stored loops of waveforms, the samples of which are played in order according to the current value of the *phase accumulator*, which matches the current time value to an appropriate sample. Frequency can be adjusted by skipping or repeating samples of the stored waveform. This approach has a very small processing time cost, as rather than performing multiple complex mathematical operations, amplitude values must only be read from memory. [11]

2.2.5 Phase Distortion

Rather than simply incrementing the phase accumulator at regular intervals, phase distortion relies on using a modulator to modify the phase accumulator, completely changing the resulting waveform compared to the stored waveform, producing a different sound. [12]

2.3 MIDI (Musical Instrument Digital Interface)

MIDI "first appeared in 1982", following an agreement between electronic musical instrument manufacturers to develop a consistent hardware and software standard for instruments.

The standard specifies inputs and outputs of MIDI devices, including the ability to network various devices together. The software specification includes timing synchronisation, multiple channel voices (including note information), alongside other features.

MIDI is commonly used in the form of controllers, often in the arrangement of a piano's keys, with additional controls, or in the form of sequencing software, usually included in a DAW. [13]

Due to its almost universal manufacturer support, MIDI is compatible with the majority of music software, across a wide range of platforms. Also, if MIDI is implemented as an input for a software synthesiser, any MIDI keyboard controller will function as originally intended.

2.4 Open Sound Control

Open Sound Control (OSC) is a protocol which allows for the control of "electronic musical instruments" [14] over a computer network, allowing for the remote control of a software synthesiser, for example.

However, since OSC is simply a protocol, it can also be used internally within a single system, with a driver program converting physical inputs to commands sent to a synthesis program.

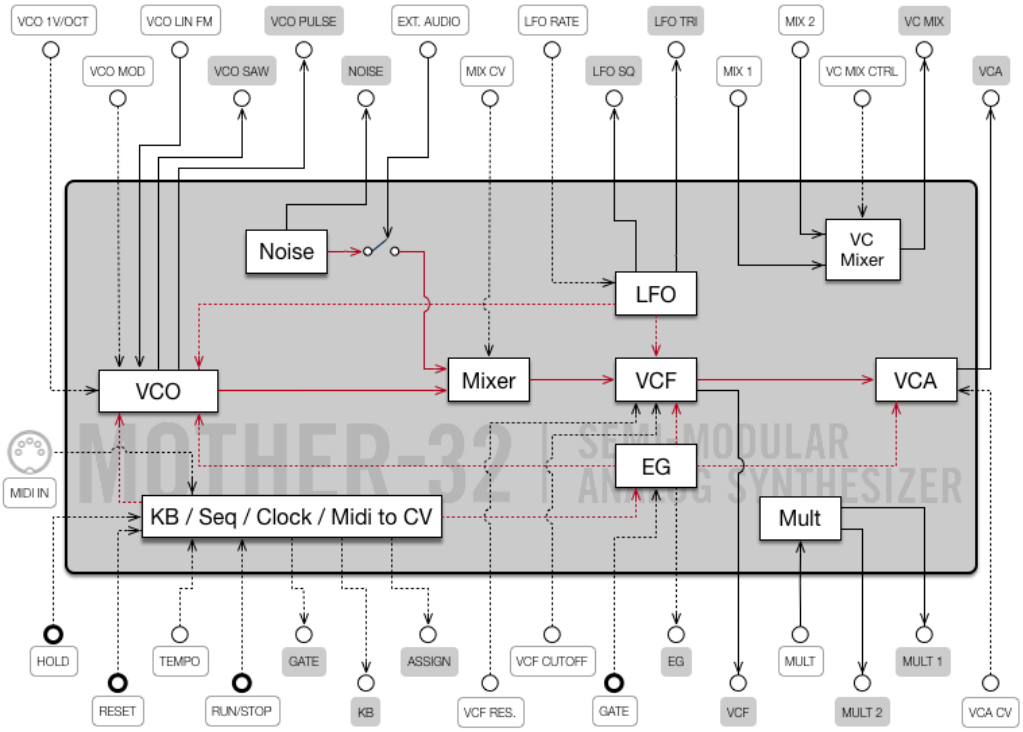


Figure 2.3: Moog Mother-32 system diagram [3]

2.5 Modular Synthesisers

Modular synthesisers consist of a series of modules, which each handles a particular audio synthesis functionality. These modules can send and receive audio signals, in the form of electricity of an analogue voltage. By connecting the inputs and outputs of various modules together, a full synthesiser can be created.

However, unlike a conventional synthesiser, which has its signal processors (such as oscillators, filters, and ring modulators) already connected in a particular, unchangeable order, modular synthesisers can combine any available module in any particular order, giving a musician or sound engineer more control over the audio produced. [2, p. 5]

Semi-modular synthesisers, such as the *Moog Mother-32* serve to bridge the gap between conventional synthesisers and modular synthesisers, by providing a single system with a series of signal processors, with a patch bay to allow the signal path to be modified, as shown in figure 2.3. [5]

2.6 Control Voltage (CV)

Modules are linked together by a series of patch cables, from a source to a destination. The source outputs an electrical current with an analogue voltage, which is in turn read by the destination.

This voltage is called a control voltage, and can be used for various purposes, including specifying pitch, a filter's cutoff, and when a parameter should be activated. [15]

2.6.1 Volts per Octave (V/Oct)

Volts per Octave is one method of using CV to control pitch. Since every Volt represents an octave, increasing a signal's voltage by 1V will increase the pitch by an octave. Increasing the voltage by a decimal value will change the note. [16]

2.6.2 Hertz per Volt (Hz/V)

Hertz per Volt is another method of pitch control via CV. Since the frequency of an oscillator would be proportional to voltage, and doubling frequency increases the pitch by an octave, to raise the pitch by an octave, the voltage must be doubled. Therefore, this mapping is more limited, as there is a smaller range of possible frequencies for the same voltage range. [16]

2.6.3 Trigger

Triggers are CV signals which control the activation of parameters, such as beginning an envelope [2, p. 97], or activating a filter. [15]

2.6.4 Gate

A gate trigger simply states if a module's parameter on or off. When the voltage is below a threshold (such as 1V), the gate signal is off, and on when it is above another threshold (such as 5V). For example, if a gate signal is converted from a MIDI input, the gate will be set to on when a key is held down, and set to off when all keys are released. [15]

2.6.5 Clock

Trigger signals can be generated regularly or rhythmically. These trigger signals are referred to as clocks, and can be used to activate parameters in time. Sequencers can be used to generate clock signals. [15]

2.7 Modules

A range of modules are commonly used in most modular synthesiser systems, with each serving a particular purpose, and various modules from different manufacturers giving different results.

2.7.1 Interfaces

Various interface modules are available which allow for external control over the modular synthesiser, by providing various outputs based on a series of inputs.

Some interfaces allow for a computer to be connected, so note outputs of a Digital Audio Workstation (DAW) can be used as inputs of the synthesiser.

Other interfaces, such as *Pittsburgh Modular's MIDI3* convert a MIDI signal to a CV, allowing for MIDI input - so a keyboard can effectively be added to the synthesiser. [17] With appropriate computer output hardware, such a module could also be used for input from a DAW.

2.7.2 Oscillator

Oscillators produce simple waveforms at a particular frequency and amplitude. Each of these waveforms has a particular shape, including square, triangle, sawtooth and sine.

The frequency will correspond to a desired note in most cases, and in modular synthesisers this is often determined by a CV input.

Various CV signals can be used to control the amplitude and frequency of an oscillator's output.

2.7.3 Low Frequency Oscillator (LFO)

LFOs are oscillators which are set to frequencies lower than the human hearing range, used to produce CV signals for inputs of other modules.

Notably, LFOs are commonly used to modulate an oscillator's amplitude, giving a "pulsing" effect.

2.7.4 Filter

"A filter is an amplifier whose gain changes with frequency." [2, p. 81] A function maps the filter's gain to frequencies, often represented as a curve. [2, pp. 81–87] There are several types of filters, as shown in figure 2.4.

In *low-pass* filters, the amplitude of the filtered signal decreases as frequency increases, from a maximum to 0dB, defined by a *slope*. The slope starts at a frequency defined as a *cut-off*. *High-pass* filters act in the opposite manner - the amplitude increases from 0dB to a maximum as frequency increases around the slope. The shape of these slopes vary with different filter designs. [2, pp. 82–83]

Band-pass filters are essentially a combination of a low-pass and a high-pass filter, where the cut-offs are centred around the same frequency. The difference between the cut-offs can also be defined as a parameter, the *width* or *pass-band*. [2, p. 84] Conversely, "a *notch filter* is the opposite of a band-pass filter" [2, p. 84], removing sound around a particular frequency, of a defined width, the *stop-band*. [2, pp. 84–85]

The cut-off frequency of a filter can be matched with the frequency of a note being played, which can serve to alter the sound characteristics of a synthesiser, such as harmonics. This process is known as *pitch tracking* or *keyboard scaling*. [2, p. 85]

The peak amplitude of a filter can be increased around the cut-off frequency, this is known as *resonance*. [2, pp. 88–86] Resonance is a characteristic associated with the *Moog Multimode Filter*. [18]

2.7.5 Ring Modulator

Ring modulators take two input signals, a *carrier* and a *modulator*. These are combined together by multiplication, producing a "metallic" sound output, with a spectrum which contains two notes, at the *sum* and *difference* of the two input frequencies. Ring modulation modules either take two input CV signals, or only one, with an internal oscillator used as a modulator. [19] [20]

2.7.6 Sampler

Samplers playback audio stored in a *soundbank*, upon being triggered by a CV signal. Playback can either be a "one-shot" or continuously looped whilst the module is receiving a gate signal. Samplers also often have the capability to change the pitch and playback rate of this sample. An example of a modular sampler is Doepfer's *A-112*. [21] The *Akai MPC2000* is a notable non-modular sampler [22], and most DAWs (Digital Audio Workstations) have sampling functionality.

Sample and Synthesis (S&S) is a technique which involves the post-processing of a sample to create a unique sound, used in a type of synthesiser which is sometimes called a *ROMpler*. [2, pp. 181–183]

2.8 Raspberry Pi

The Raspberry Pi is a cheap, low-power ARM-based computer that is the size of a credit card. These properties are highly beneficial for creating small electronic devices.

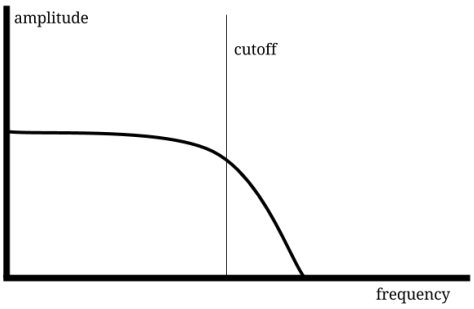
The Generation 1 Model B has 17 GPIO pins, which use electrical currents as digital input or output, with high voltages representing 1 and low voltages as 0. The values of these pins can be set and read by programs using an API, allowing the GPIO to affect the control flow of a program. The polling rate of the GPIO is up to 70kHz. [23]

Raspberry Pis support several specific distributions of the GNU/Linux operating system, offering a wide range of program support.

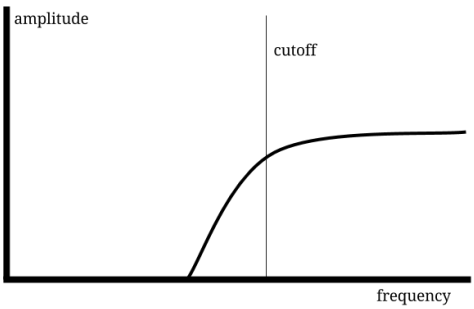
2.9 Arduino

"Arduino is an open-source platform used for building electronics projects." [24] Arduino computers can therefore be compared to the Raspberry Pi, which is also frequently used in electronics projects, and both systems have input/output pins.

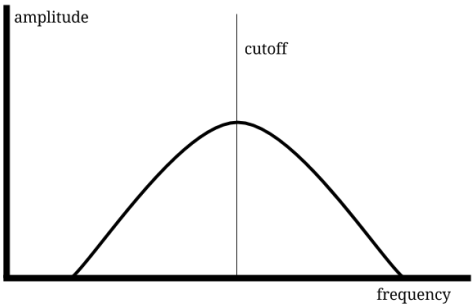
However, the systems have some differences - the Arduino does not have a traditional operating system, and is simply a microcontroller programmed using a C-like language. [24]



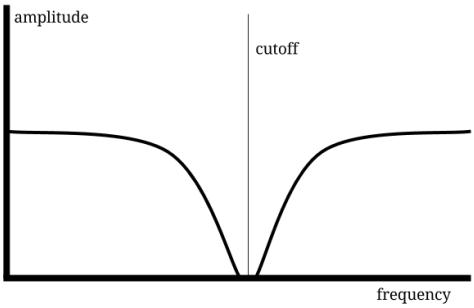
(a) Lowpass filter



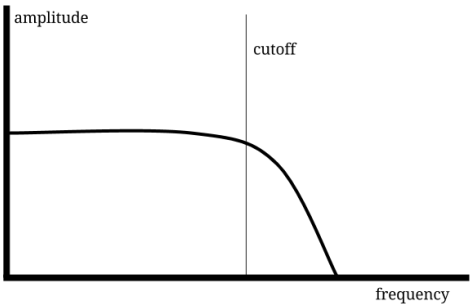
(b) Highpass filter



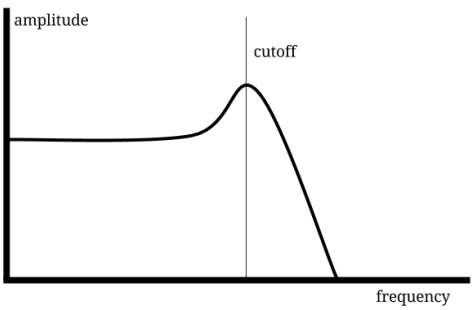
(c) Bandpass filter



(d) Notch / Bandstop filter



(e) Lowpass, with low resonance



(f) Lowpass, with high resonance

Figure 2.4: Various filters

Notably, some Arduino systems have analogue pins [25], unlike the Raspberry Pi, allowing for analogue voltage to be read, such as input from a potentiometer or thermistor. Arduinos can also send signals to other devices via "USB or over Ethernet" [25], including the Raspberry Pi, so an Arduino can effectively act as an analogue I/O header for a Raspberry Pi.

2.10 MCP3008

The MCP3008 [26] is an integrated circuit ADC (Analogue-to-Digital converter), which converts up to eight analogue voltage inputs to a digital signal, which can be read by a computer, such as a Raspberry Pi with a simple driver program.

Aside from positive and ground pins, the board only occupies four GPIO pins on the Raspberry Pi itself, to read which analogue input should be read, and to transmit a 10-bit representations of this input, using an output signal on one pin (Data Out) which changes to transmit this value upon reading the a signal on another (Clock).

Notably, the value of a potentiometer can be read by simply connecting its terminals to ground and positive power, and the wiper terminal to the analogue input pin.

2.11 DSP (Digital Signal Processing)

Aside from analogue systems (such as traditional synthesis modules), sound can be processed by either DSP-specific microcontrollers [27, p. 352] or software running on a CPU interfacing with a soundcard (such as a *Creative SoundBlaster*). [27, p. 355]

An analogue audio waveform input can be converted to a digital waveform representation using an *Analogue-to-Digital Conversion (ADC)* system. [7, p. 29]

However, in order to be manipulated by computers, sound must be represented as digital signals (which can be stored and streamed as a series of bits). [7, p. 27] Notably, digital signals can be expressed either as a waveform (a representation of physical soundwaves), or as a spectrum of a waveform, which is essentially a map of the amplitudes of individual frequencies against time. [7, p. 28]

If a spectrum representation is used, manipulation is fairly simple. For example, to implement a low-pass filter, a mathematical function taking the cut-off as a parameter would be applied to the spectrum, suppressing the amplitudes of frequencies above the cut-off according to the defined curve.

The *Fourier transformation* allows for a spectrum to be calculated for any given waveform, which plots amplitudes and phases at all frequencies over time. [7, p. 29] No information is lost when converting from a waveform to a spectrum and vice-versa, provided both amplitude and phase information is conserved. [7, p. 148]

After a digital signal has been processed, it can be converted back to an analogue form using a *Digital-to-Analogue Converter (DAC)*, to be output via an amplifier to a speaker. [7, p. 29] This can be done in realtime (such as using a digital synthesiser for a live performance), or on a stored audio file, such as an *MP3*, or a *Compact Disc (CD)*.

2.12 Audio Programming

It is possible to synthesise audio using what is effectively a programming language.

There exist several software solutions which fulfil this purpose.

2.12.1 Csound

Csound is a digital signal processing programming language oriented towards music synthesis.

A programmer creates an *orchestra*, which defines a series of *instruments* and which DSP functions are applied to generate sound. A series of instructions of what notes to play at given time periods is declared as a *score*, which also specifies which orchestras and instruments should be used.

Also, the Raspberry Pi and other ARM-based microcomputers are supported by Csound, with various examples featured in the Csound journal, including using a network application to control sound output.

Csound features an API written in C, with wrappers for Java and Python. This API allows for Csound functionality to be implemented using these programming languages, providing the ability to manipulate sounds with particular inputs. However, if the "instrument" is modified, it must be recompiled before the audio output is changed.

Concise documentation is available [28], featuring full explanations and examples of every opcode and operator, plus various example orchestras and scores alongside configuration guides.

MIDI and Open Sound Control are fully supported by Csound. To integrate MIDI, the MIDI controllers to be used are simply defined, and the function *cpsmidi* returns the active MIDI note "expressed in cycles-per-second units" [29], the frequency of such a note in hertz, which can be used as the frequency of an oscillator.

2.12.2 Pyo

Pyo is a Python DSP module, with native Python syntax. Pyo also has a considerable benefit that the audio processing chain can be modified "in real time through the interpreter" [30], so no recompilation is necessary when a patch is modified, unlike Csound.

Audio file loading is also available, so a ROMpler (sample-based synthesiser) can be implemented using Pyo.

The module is compatible with both Python 2.7 and Python 3 [31], and is fully compatible with the Raspberry Pi. [32]

Pyo also supports OSC and "the MIDI protocol for generating sound events and controlling process parameters" [30], allowing for interaction via a MIDI controller or another program.

2.12.3 SuperCollider

SuperCollider is a audio synthesis platform which consists of *scsynth*, "a real time audio server" [33] which executes synthesis and signal processing. Signal processing is determined

using *sclang*, a programming language which "controls scsynth via Open Sound Control." [33]

MIDI is fully supported by SuperCollider [33], and SuperCollider is compatible with the Raspberry Pi. [34]

2.12.4 Pure Data

Pure Data is a visual programming language, with cross-platform capabilities, including Raspberry Pi support.

Particularly, Pure Data is capable of audio processing and synthesis. An instrument can be created by linking *objects* together with a series of *cords*, where "each object performs a specific task" [35], such as "mathematical operations" or specific audio processing.

pdwiringPi, a Raspberry Pi GPIO plugin based upon *WiringPi* is also available, yet there is only limited documentation available.

Whilst the visual programming approach makes the creation of basic instruments simple, creating a potentially large and complex program, which processes multiple inputs could become more complex than using a conventional object oriented programming language.

2.12.5 ChucK

ChucK is a cross-platform, open-source audio programming language, which executes under its own virtual machine. Notably, ChucK supports *"the ability to add and modify code on-the-fly"* [36]. This realtime code modification allows for a synthesiser's properties to be changed without the need for recompilation.

A third-party Python wrapper for ChucK exists, which can be used with the Raspberry Pi GPIO API to allow for sound modification based upon GPIO input. However, this wrapper is less commonly used than the CSound Python wrapper, and is only maintained by one developer, therefore it has less documentation in comparison.

2.12.6 Sonic Pi

Sonic Pi is a music programming package designed for *"live coding"*, a type of live musical performance which involves modifying a program, to controls a sound output to be updated on the start of a bar, often with a display for the audience to view the source code.

As its name suggests, Sonic Pi is compatible with the Raspberry Pi. [37] Sonic Pi is targeted at programming education at Key Stage 3 level, and is therefore simple to learn and use. [38]

However, Sonic Pi operates as a single GUI environment, and external control is fairly limited. Whilst MIDI is supported, control via GPIO is not currently officially supported by Sonic Pi.

2.13 Case Manufacturing

A synthesiser's electronics are usually housed within a casing, with a few notable exceptions, such as the *Teenage Engineering Pocket Operator* [39] series. This casing prevents the electronics from being damaged.

Computer Aided Design (CAD) is an effective process for designing the case, as it allows for precise measurements (for mounting components), and can be sent to an manufacturing machine for production.

2.13.1 3D Printing

Additive manufacturing, otherwise known as 3D printing, is a manufacturing process which involves building up an object by building up material "layer by layer". [40]

This process allows for complex three-dimensional objects to be created from a design produced in computer software. Notably, modern 3D printing approaches are capable of creating a detailed physical model of a human patient's heart, with no intrusion into the body. [41]

However, considerable post-processing on a conventional CAD file must be performed in order to create a Standard Tessellation Language (STL) file, which can be read by a 3D printer. [42] Furthermore, 3D printing is an expensive process, requiring "specialised equipment and consumables". [43]

2.13.2 Laser Cutting

Laser cutting is a manufacturing process in which a high power laser cuts through a material via vapourisation due to the laser's radiation. [44]

There are several benefits to laser cutting wood. In particular, resulting cuts are smooth compared to using conventional tools. [44]

Additionally, laser cuts are accurate, as lasers are often controlled by a CNC (Computer Numerical Control) machine, reading a cutting path from a CAD file with precise inputs. [45] [46] The ability to use CAD also allows for effective visualisation of the prototyping process.

2.14 Electronics

2.14.1 Printed Circuit Boards

Printed Circuit Boards (PCBs) are circuit boards with copper tracks specific to a particular electronic circuit. The copper tracks are often formed by using a process of chemical etching. [47] These tracks form a complete circuit once populated with soldered components, with no additional joins required.

The circuit image used to isolate etching region can be designed using a software suite such as *gEDA* (GPL Electronic Design Automation Tools). [48]

However, PCBs are costly to manufacture as one-offs, and designing the tracks correctly can take a considerable amount of time [49], especially if the tracks have to be optimised to make the PCB as small as possible.

2.14.2 Stripboard

Stripboard is a simple circuit board consisting of a series of straight, perforated copper tracks mounted on a board. [50] Components and "fly-wires" can be soldered to these tracks to form circuits. The tracks can also be cut to disconnect two terminals from each other. [49]

This simple, generic design makes stripboard a very cheap option for one-off circuit manufacturing. [49]

Since components and "fly-wire" connections are soldered directly to the board, no design materials (such as etching graphics used for PCBs) need to be prepared beforehand, other than perhaps simple circuit schematics to use as a reference.

However, stripboard is not particularly suited to complex circuits with many components, as tracks have to be cut to isolate components, and fly-wires must be connected to join different parts of the circuit together. [49] [49]

2.15 Summary

Modular synthesisers provide more artistic freedom than conventional synthesisers. However, they are also much larger, and therefore less suitable for live performances due to transport issues, aside from the considerable cost of buying separate modules compared to a single synthesiser. Semi-modular systems are an effective compromise between these two design paradigms.

MIDI support is highly beneficial for a synthesiser, as it allows for a wide range of hardware to control the sound produced, either a simple keyboard, or a pre-composed score running on a computer can be used for note input.

The Raspberry Pi is an appropriate computer to use for a software-based synthesiser, and its GPIO pins would allow for a simulation of a semi-modular patch bay. An Arduino microcontroller could also be used to send analogue input to the Raspberry Pi, allowing for the manipulation of parameters using potentiometers.

Whilst various audio programming languages exist, Csound and Pyo appear to be the most suitable options for writing a software synthesiser, due to their extensive documentation and cross-platform support. Also, due to considerable Raspberry Pi integration and audio library support, Python is an excellent option for programming the system's logic.

Chapter 3

Requirements and Analysis

3.1 Aims

There are several aims for the final synthesiser of this project:

- *Portability* - the synthesiser should be in a single casing, preventing damage to electronic internal components, and should be as small as possible.
- *Semi-Modular* - the signal path should be able to be modified, allowing for more control over the produced sound.
- *Parametric* - parameters of individual signal processing systems should be controlled by analogue inputs.
- *Freedom* - the final software implementation should only use free software (such as GNU/Linux), with no proprietary software necessary to function. This would allow for future customisation of the software by end-users.

3.2 System Plan

The system should simulate a semi-modular synthesiser, as a truly modular synthesiser would require much more hardware to implement, such as multiple Raspberry Pi computers and high quality audio interfaces for each of them. However, having a customisable signal path would be beneficial to sound design.

The Raspberry Pi's GPIO is limited to digital input and output only, so actual analogue audio signals cannot be transferred over them for manipulation by various software modules.

However, the GPIO can still be used to simulate patching. If every 'module' has an input and an output GPIO pin, and another two pins are used as a global *start* (always set to 1) and an *end*(the final pin in a chain), a script can determine the signal order. Output pins would be connected to input pins by the user (with the first output as the start pin) to link modules together, until the end pin is connected. If an output pin is connected to an input,

the input will read the output's value. The script would read the input pin of each module, and if the value of 1 is read, the module is added to the next point in the signal chain, and its output pin is set to 1. This process would be repeated until the end pin reads a value of 1, and the new signal chain will be set. This process should update in a short amount of time, due to the Raspberry Pi GPIO's high polling rate of up to 70kHz. [23]

Due to a lack of onboard analogue input on the Raspberry Pi, an MCP3008 could be used to receive input from up to eight potentiometers, allowing for various parameters of different modules to be controlled. Additionally, the MCP3008 only needs to be connected to four of the Raspberry Pi's GPIO pins (excluding 3.3V power and ground).

Python has considerable Raspberry Pi support, including GPIO interfacing, and should therefore be used for programming the system. A Python compatible audio processing library, such as Pyo or Csound will be used for synthesis and signal processing. A simple prototype synthesiser will be developed using each library in order to determine which is the most suitable for the system.

3.3 Stakeholders

There are multiple potential stakeholders for this project, and their needs were noted when making the project's aims, and should continue to be considered during development.

- *Musicians* - electronic musicians would benefit from more synthesisers, for both studio production and live performances. Low cost hardware which is compatible with existing controllers would address concerns of budget-conscious artists, and a highly configurable and modifiable system would be appealing to those who wish to experiment.

- *Teachers and Students* - a synthesiser based on a Raspberry Pi would be a useful teaching tool. A synthesiser could be used to teach music theory and sound engineering, sound processing software running on the Raspberry Pi computer would be practical for computer science education, and the electronics in the system and a casing to house them would be a valuable project for design and technology students.

- *The "Maker" Community* - "Makers" are electronics hobbyists, generally with an interest in creating devices with embedded computers. An easily modifiable Raspberry Pi synthesiser could be an interesting project. Providing a high level of freedom over software and hardware used in this synthesiser would allow for makers to create similar projects.

3.4 Evaluation Plan

Due to the subjective nature of music and instruments, this synthesiser should be evaluated by conducting a survey of various anonymous participants, after they have used the synthesiser.

3.4.1 Interaction Session

The participants would first be given a brief explanation of how modules can be patched together, and what each module does.

3.4.2 Survey

The survey itself will consist of several sections.

The participants will state their past experience in both electronic music (including other modular synthesisers) and music performance in general. This will serve to determine how varying degrees of relevant musical experience of various participants influence their opinions of the synthesiser.

In order to evaluate the design of the synthesiser, the participants will be asked if they agree or disagree with various statements related to several principles of design [51]. These statements include:

- *"It was clear how modules could be patched together."* - evaluates the clarity of the synthesiser's design.
- *"I felt that my actions on the synthesiser directly influenced the sound generated."* - evaluates how much 'control' the participant feels that they have over the synthesiser.
- *"The layout of the modules made it easy to patch them together, and access their knobs / buttons."* - evaluates the organisation of the synthesiser's front panel.
- *"I could easily start to patch modules together to design a sound, when no patches were originally connected."* - evaluates how effectively a user can produce a result from the "zero state" [51] of the synthesiser, where no patches are connected.

Similar subjective statements will also be used to evaluate the participant's opinions on the utility of the synthesiser, and other general features, including:

- "I like the aesthetic design of the synthesiser." - evaluates the appeal of the synthesiser's appearance.
- "I like the sounds that could be produced with the synthesiser." - evaluates the synthesis capabilities of the system.
- "I understood how to patch modules together to make a particular sound." - evaluates the accessibility and simplicity of the synthesiser.
- "I would like to use the synthesiser in future performances, production, or jam sessions." - evaluates the synthesiser's overall utility as a musical instrument.
- "I would be interested in using publicly released source code of the synthesiser to either modify it, or create my own." - evaluates interest in using the synthesiser's free software in personal projects.

- "I believe the synthesiser would be an effective teaching tool for programming, music, or both." - evaluates the suitability of the synthesiser as an educational resource.

3.4.3 Ethics Approval

This study is required to follow the University of Sheffield's "Ethics Policy Governing Research Involving Human Participants, Personal Data and Human Tissue" [52], which ensures that all research at the University is ethical, and of minimal risk.

In particular, the survey was constructed to ensure that the data gathered was completely anonymous, therefore none of the participants' identities can be linked to their responses. Additionally, participants are to be informed of the purpose of the study, and the extent of their contributions. Various other ethical concerns were also considered, as shown in the following section.

3.5 Ethical, Professional, and Legal Concerns

When using sampling, the importance of copyright law must be considered, as sampling copyrighted audio is a breach. This can be easily resolved by only using public-domain audio, which could be licensed under the "GNU General Public License" or an appropriate "Creative Commons" licence. [53]

Replicating synthesisers and components (such as the Moog resonant filter) could be regarded as copyright infringement. However, many third-party software emulators for iconic synthesisers exist, and are widely used. Notably, *Mono/Fury* is a popular freeware emulator of the *Korg Mono/Poly* [54], and hasn't faced legal issues, despite the presence of an official Korg software Mono/Poly [55]. This is likely because Mono/Fury isn't directly copying the electronics of the Mono/Poly, nor the code of the official software version, simply making a logical approximation of such circuitry instead.

Using samples without permission could also be regarded as unethical and unprofessional.

3.6 Software Licensing

Software can be legally licensed in various different ways.

Popular commercial software is often released under a "proprietary" licence, under which the developer or publisher retains all rights of the software and its source code. However, proprietary software therefore cannot be modified by the end users, restricting its functionality, and its freedom. [56]

As a key aim of this project is to allow for freedom, where future users of the software are able to make modifications to the program, an appropriate licence should be used. The "GNU General Public License" (GPLv3) offers such freedoms, and ensures that derivatives of the software maintain this freedom. [53]

Chapter 4

Design

4.1 Prototypes

In order to find which audio programming language was most suitable for the project, prototypes were written using both Csound and Pyo. These prototypes ran on GNU/Linux, with Python3 (using the wrapper in the case of Csound), on an x86 computer with a connected MIDI keyboard controller.

The prototypes both had the same aim, to create a simple oscillator (with MIDI note input to determine frequency), and add a simple Moog-style resonant filter.

4.1.1 Csound

Whilst the Csound wrapper provides Python functionality, most of is generally limited to functions, such as starting the Csound server, and loading orchestras (sound processing programs). These orchestras are written in the Csound language, and stored as strings.

In order to use signal processors in an arbitrary order, these strings would have to be combined dynamically in Python, which would be a less ideal approach than using a truly Python-based library, and using object-oriented principles.

4.1.2 Pyo

The Pyo module is accessed entirely using native Python syntax. A GUI is available for the Pyo server, which can display the waveforms and spectra of particular signals, as well as providing variable input.

Pyo offers a wide range of premade oscillator and DSP functions (including a SuperSaw and Moog resonant filters), with simple numeric parameters, which can be controlled via an external input (such as a value read from an MCP3008 ADC) or the server's GUI.

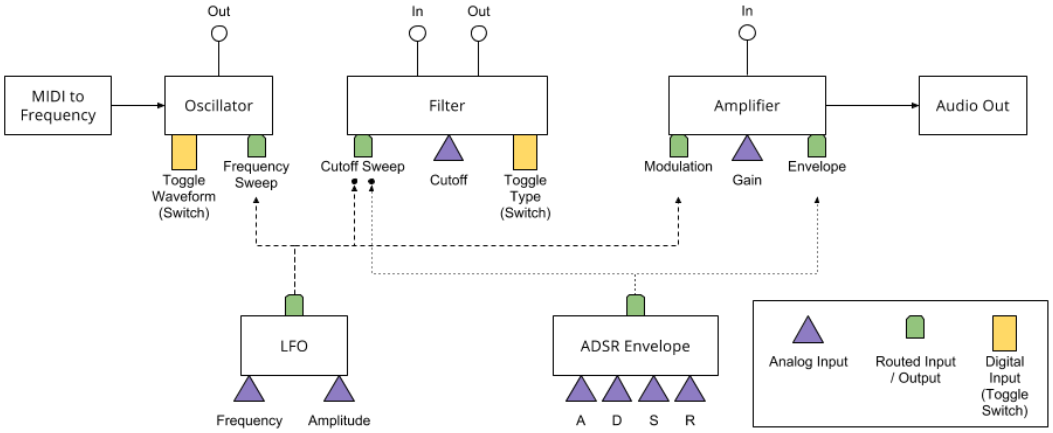


Figure 4.1: A system diagram of the synthesiser.

4.1.3 Results

Developing the prototypes showed that Pyo would be preferable to Csound, mostly due to the fact that Pyo can be written using Python syntax and easily accessible parameters, whilst Csound required for strings to be concatenated with a complex structure to set variables correctly.

The prototypes also showed that both the official Pyo documentation [57] and community resources [58] would be adequate to write a synthesiser using the module.

In addition, other projects have successfully created synthesisers [59] and other audio processing systems [60] using Pyo on a Raspberry Pi. Therefore, these existing free-software projects can be used as references to solve potential issues encountered whilst implementing the software.

4.2 System Diagram

In order to plan how modules and signal patching should be implemented in software, a system diagram was created (figure 4.1), showing possible patches between modules.

The diagram also shows which inputs (both analogue and digital) are required for each module, allowing for the maximum possible number of inputs to be used in the synthesiser, providing as much control over the sounds produced by the final design as possible.

This system diagram will be provided to evaluation participants, as a reference of the capabilities and limitations of the patching system, preventing potential confusion when an invalid patch results in no audio output, or unexpected behaviour.

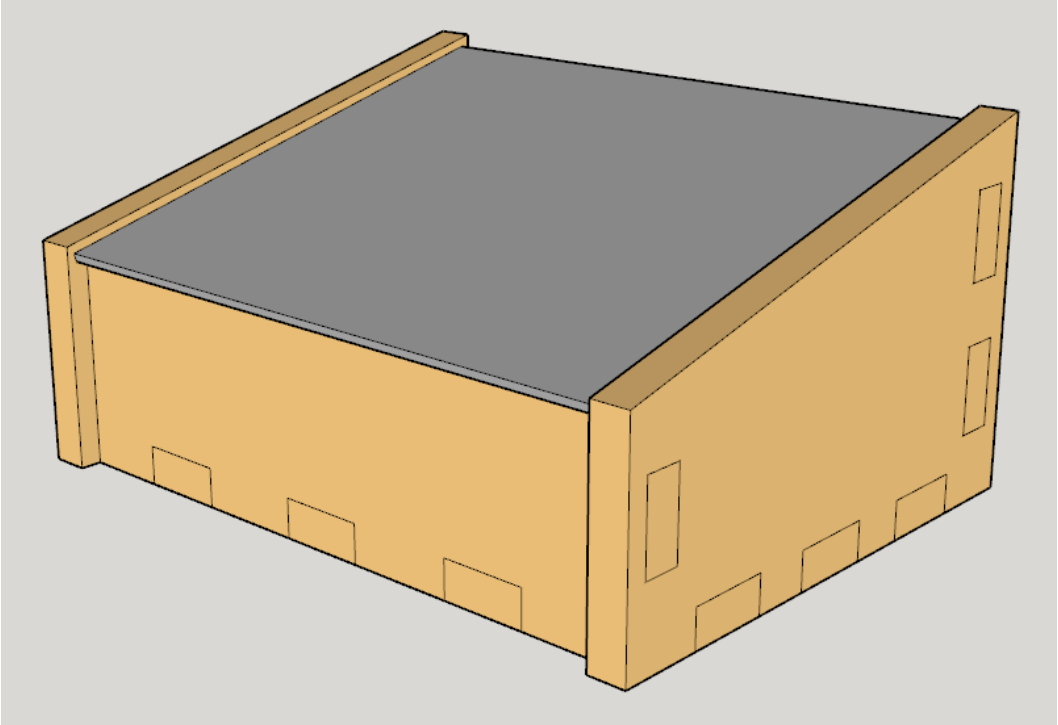


Figure 4.2: A 3D model of the prototype case.

4.3 Case Design

Computer Aided Design (CAD) was used throughout the case design process, as it allows for immediate visual feedback whilst designing.

SketchUp Make [61] was used to create a 3D model of the case (figure 4.2), a process which was simplified, as the software allows for the input of accurate measurements, such as the thickness of the materials to be used.

This 3D model was then converted to 2D DXF drawings (figure 4.3), which can be read by a laser cutter to precisely machine panels from materials.

This process involved manually taking measurements from the SketchUp model, and using them to draw the according panels using LibreCAD [62] (a free, open-source, multi-platform CAD program), alongside AutoCAD [63] (a proprietary CAD program, with more advanced features) for later modifications, such as labels.

In addition, two more drawings were made for cutouts of the top panel, including labels and mounting holes (figure 4.4), and a rear panel for a USB MIDI device and a micro-USB power input (figure 4.5).

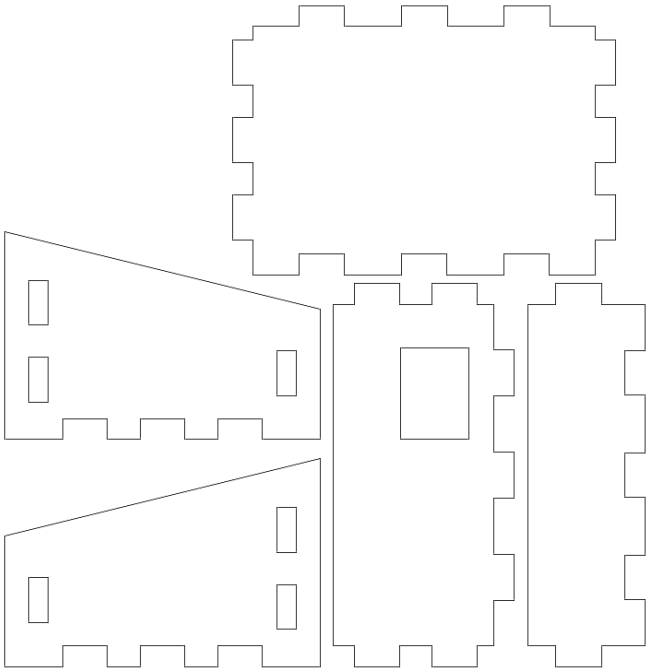


Figure 4.3: 2D drawings of the plywood panel cutouts.

4.4 GPIO Pinout

The Raspberry Pi's GPIO pins can be accessed by one of two different pin layouts (pinouts), which must be set when initialising the Python program used to access them.

One of these numbering systems is simply known as "BCM", where the pin numbers are the same as those of the channels on the system's Broadcom SOC (System on a Chip). However, this numbering system varies across different Raspberry Pi board revisions, and therefore will have reduced portability across different systems. [64]

Conversely, the other numbering system, referred to as "BOARD" numbering is based on the physical pin order on the Raspberry Pi itself, as shown in figure 4.6. This system has two key advantages:

- The pin numbers are simply in order, and can be counted to find which pin will need to be connected when building the system, compared to the more complex "BCM" numbering, which requires the user to use a reference. [4]
- As revisions of the Raspberry Pi retain the same physical pinout, GPIO programs are portable across different board revisions. Whilst some later revisions have more GPIO pins, the first pins are the same as those on the original boards. [64]

Table 4.1 shows the planned inputs and outputs assigned to each GPIO pin. Some of these pins are power rails (3.3V, 5V, and ground), which can be used to connect electrical

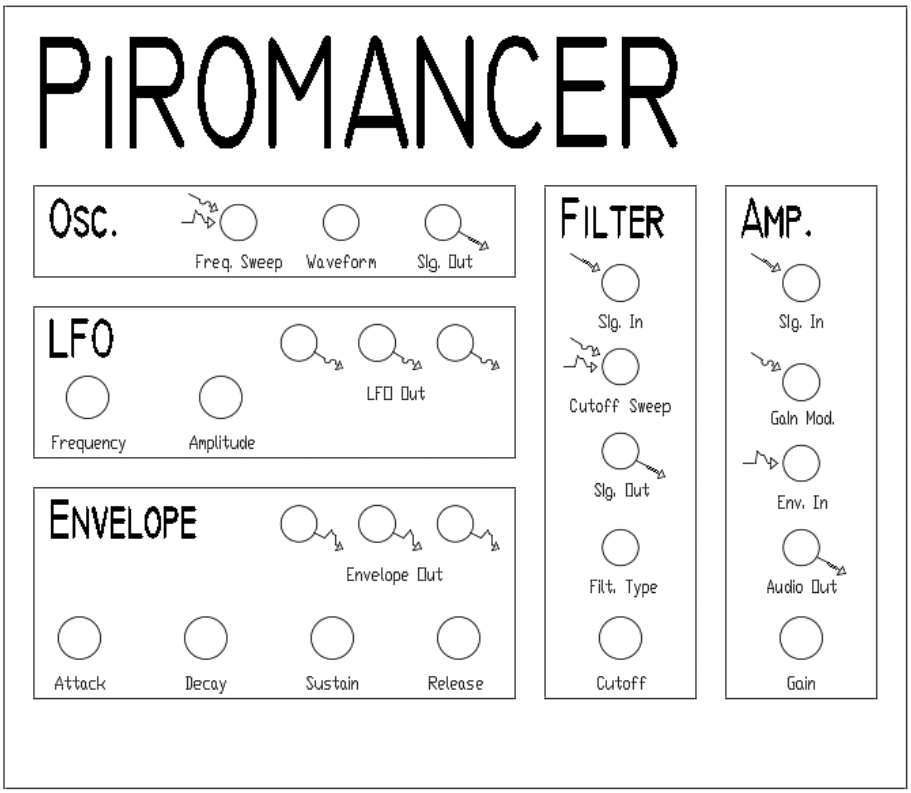


Figure 4.4: Top panel 2D drawing.

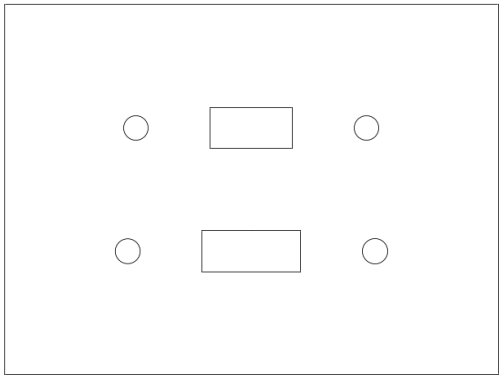


Figure 4.5: Rear panel 2D drawing.

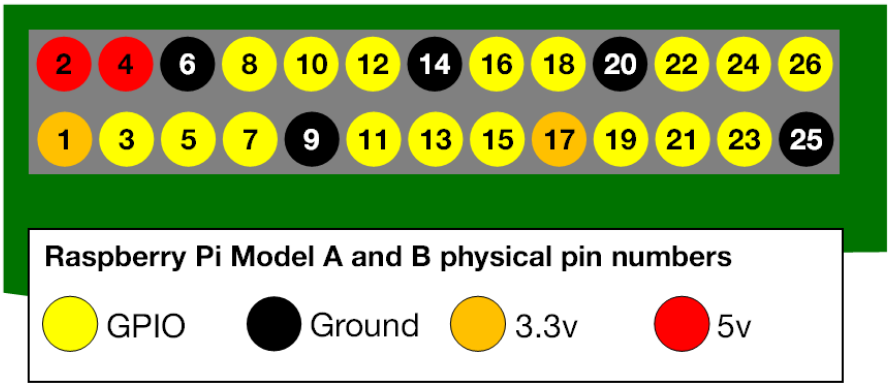


Figure 4.6: GPIO pinout (physical numbering). [4]

components to the Raspberry Pi.

Each patching port will be connected to a pin, alongside the MCP3008's input and output pins. Additionally, momentary switches can be connected to a GPIO pin and the ground rail. [65]

4.5 Circuit

As the MCP3008 DAC integrated circuit must be connected to the Raspberry Pi, and several potentiometers to be used as analogue inputs, a circuit must be built.

4.5.1 Breadboarding

Based on the MCP3008 circuit example written by Michael Sklar for Adafruit Industries [26], a physical circuit prototype was created using a breadboard and jumper cables (figure 4.7). Building a breadboard prototype allows for any potential issues to be found before spending time manufacturing a circuit board, and to quickly test new ideas by simply changing cable connections. [49]

The example driver program [66] was modified to simply print the value read by the MCP3008, rather than change the output volume of the Raspberry Pi.

A potentiometer was connected to the circuit, with the outer pins connected to each of the power rails, and its centre pin connected to one of the MCP3008's analogue inputs.

Since multiple potentiometers can be connected to the power rails in parallel, using more potentiometers would have no impact on the output voltage of each potentiometer.

After connecting the Raspberry Pi's GPIO pins to the appropriate digital pins on the MCP3008, the modified program was run on the Raspberry Pi. As the potentiometer was turned, the output number changed accordingly, ranging from a minimum of 0 to a maximum 1024, when the potentiometer was turned to each of its limits.

Table 4.1: GPIO connections. (Connections in bold are power rails built into the Raspberry Pi.)

Pin	Connection
1	**3.3V**
2	**5V** - Unused
3	[Osc.] Signal Out
4	**5V** - Unused
5	[Env.] Signal Out
6	**Ground**
7	[LFO] Signal Out
8	[Filt.] Signal In
9	**Ground**
10	[Filt.] Signal Out
11	[Osc.] Change Waveform (Button)
12	[Amp.] Signal In
13	[Filt.] Change Filter Type (Button)
14	**Ground**
15	[Amp.] Envelope In
16	-
17	**3.3V**
18	[MCP3008] CS
19	[Amp.] Gain Modulation (LFO In)
20	**Ground**
21	[Filt.] Cutoff (Env. / LFO In)
22	[MCP3008] Digital In
23	[Osc.] Frequency Sweep (Env. / LFO In)
24	[MCP3008] Digital Out
25	**Ground**
26	[MCP3008] Clock

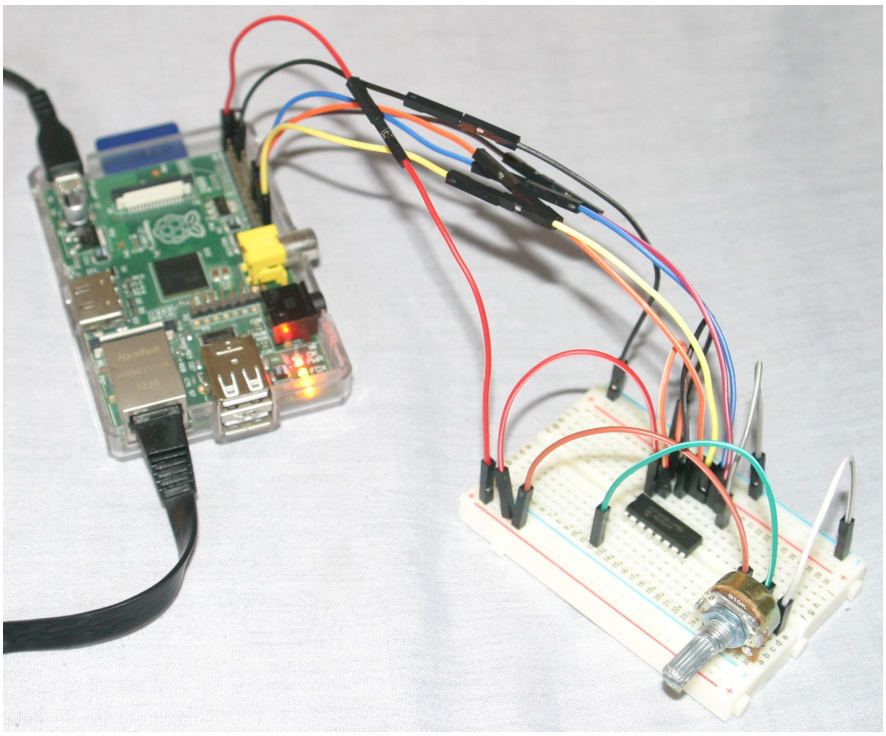


Figure 4.7: The breadboard prototype.

Additionally, the test potentiometer was connected to each of the analogue input pins, in order to test that they each functioned correctly. This breadboarding process confirmed that the MCP3008 and the driver program functioned correctly.

4.5.2 Schematics

The breadboard prototype could then be converted to a 2D schematic, using gEDA's (GPL EDA) schematic program. [48]

This schematic (figure 4.8) was extended to include all eight potentiometers required for the synthesiser (four for the attack, decay, sustain, and release parameters of the envelope; two for the frequency and amplitude of the LFO; one for the filter's cutoff frequency; and one for the amplifier's gain).

Additionally, the two required switch inputs were included on the schematic, as they would also use the same ground power rail, and therefore should be included in the circuit.

4.6 Software

4.6.1 Architecture

The system will adopt an object-oriented design, with three classes, each performing a specific role:

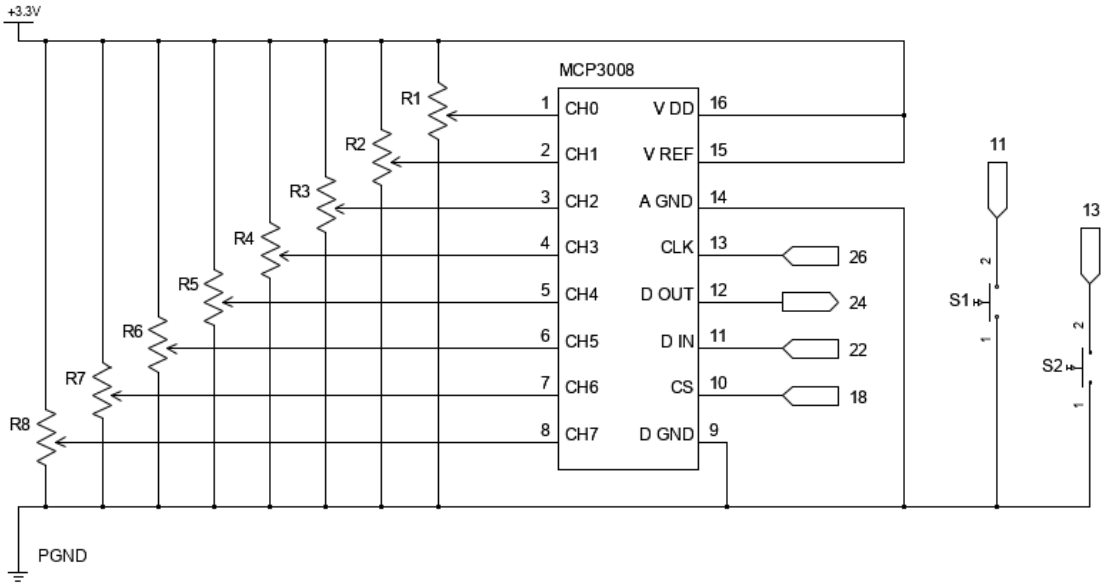


Figure 4.8: 2D circuit schematic.

- patcher.py - Finds which patchbay pins are connected together, and determines the order in which modules are connected together.
- mcpaccess.py - Reads integer values of the MCP3008's eight analogue potentiometer inputs.
- buttons.py - provides access to the push-to-make switches connected to the GPIO, and modifies the variables which these switches should control.
- pyohost.py - Initialises and accesses other classes, as well as the Pyo server. Responsible for audio processing, depending on values read by the other classes. Essentially the 'main' class.

Figure 4.9 shows this object-oriented architecture, where instances of mcpaccess and patcher are created and accessed by pyohost.

4.6.2 Algorithm Design

In order to determine the patchbay's signal path, a recursive algorithm inspired by functional programming paradigms can be employed.

At each step, if a module's signal input GPIO pin reads a high value, the module is added to the signal path queue, as this must be connected to a high pin. This module's signal output pin is then set to high, whilst the other module's output pins are set to low.

This function would be called recursively until the amplifier module is reached, in which case the queue is returned, or no next input pin has a high reading, where an empty queue is returned instead, as no audio should be produced, since the amplifier never receives a signal.

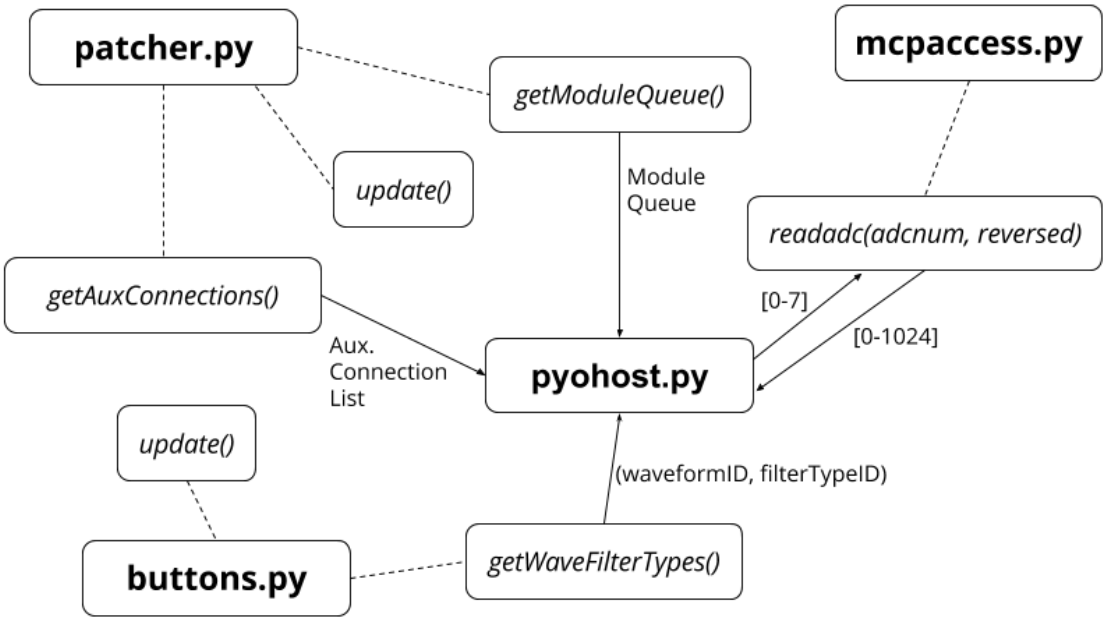


Figure 4.9: Software architecture diagram. The italic boxes represent functions of the classes which pyohost calls. The arrows show input parameters and return values.

For the first step, the oscillator module is assumed to be the input, as it will always be the first module in the signal path for this synthesiser, as it is the only (non-control) signal generator.

Auxiliary connections, such as connecting the envelope or LFO modules to other modules, simply have two states (connected or disconnected), and therefore do not need to be in order, and can simply be added to a list if they are present.

A similar recursive algorithm is used to compute the signal processing steps required to simulate the modules. The first module is popped off the signal queue and an according signal processing step is set to be applied to the Pyo audio signal, which passed into the function, alongside the remaining queue.

Since the first module is the oscillator, the base step is a simple oscillator, which is passed into the recursive step, along with the remaining signal queue.

This recursion continues until the queue is empty (in which case no signal is generated), or the amplifier module is reached, where the Pyo audio signal is set to output.

In the system's main logic loop, if there is a detected change in the signal path queue, auxiliary connections, MCP3008 pin values, or identifiers controlled by button presses, the signal processing steps required for the audio signal will be recomputed, Pyo's signal processor will be updated, and a different sound will be produced accordingly.

These changes will be detected by storing a copy of these tracked values, before updating them. If the copied and updated values are different, then a change has occurred.

Chapter 5

Implementation and Testing

5.1 Case Manufacture

In order to prepare the CAD drawings for the laser cutting machine, the schematics were modified so that the panels were effectively 0.15mm thicker. This was performed to account for the width of the laser, which is 0.3mm. Therefore, the cut panels would be the same size as the original design, as the added 0.15mm radius of the laser would be removed during the cutting process.

The main case panels were cut by placing laser safe birch plywood onto the cutting tray, aligning the laser, and transferring the appropriate CAD file to the cutting machine. Once the laser cutter had finished, the panels were carefully removed from the wood. However, in some areas, the laser did not completely penetrate the material, so some panels had to be prised out in these areas, causing some splintering and chipping in the more fragile areas. These edges were sanded to remove the splinters, and to make the chipped edges more consistent.

The same process was repeated with acrylic sheet for the rear USB port panel and the front patchbay panel. However, the patchbay panel features some elements which should be engraved into the plastic, rather than cut out, such as text and the module borders. These were marked in CAD software with a different colour, and the machine was configured to move the laser across these lines at a higher speed. This prevents the laser from fully penetrating the sheet in these areas. Since the laser fully penetrated the sheets where cuts should be made, no additional steps were required to prepare these panels, other than simply cleaning off residue.

To assemble the case, the wooden panels were simply slotted together via the appropriate joins, as the friction between the panels was sufficient to hold them together. The sides of some joins were filed back, where they would otherwise be too tight to fit together.

The rear USB panel was attached to the case with adhesive, and held in place with a clamp until secured (figure 5.1).

In order to mount the patchbay panel to the case, four holes were drilled into the top of the casing, with the same depth, perpendicular to how the patchbay panel would be mounted. Standoffs were inserted into each of these holes, and held in place with an expanding adhesive.

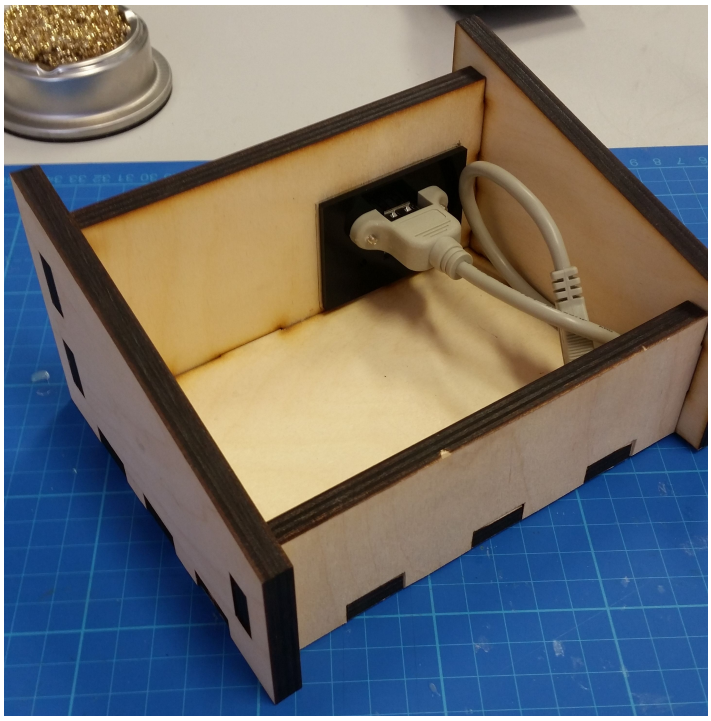


Figure 5.1: The casing, with the rear USB panel mounted.

The patchbay panel was then placed on top of these standoffs, and their locations were marked onto it. These markings were then drilled, allowing the panel to be mounted onto the standoffs with screws (figure 5.2), with the ability to be removed for access.

5.2 Electronics

The main circuit for the system was built using stripboard.

In order to prevent heat produced from the soldering iron from damaging the MCP3008 chip, an integrated circuit socket was soldered to the stripboard instead (figure 5.3). The MCP3008 was then inserted into this socket once the soldering was complete.

As this socket was soldered to allow for each strip on the stripboard to be connected to an individual pin, to prevent two orthogonal pins (which would be on the same strip) from being connected together, a countersinking drill bit was used to break the strip between them (figure 5.4).

Where wires were to be soldered to the board, small holes were drilled nearby, on strips which were not used for connections, or at the ends of strips, where connections will not be made. The wires can then be routed through these holes, before being soldered (figure 5.4). This prevents the solder join from breaking if the wires are strained, as the side wire's sheathing will simply be pulled against the board instead, rather than a thin area of the wire's core, which is weaker.

Figure 5.2: The front panel mount, using a standoff and a screw.

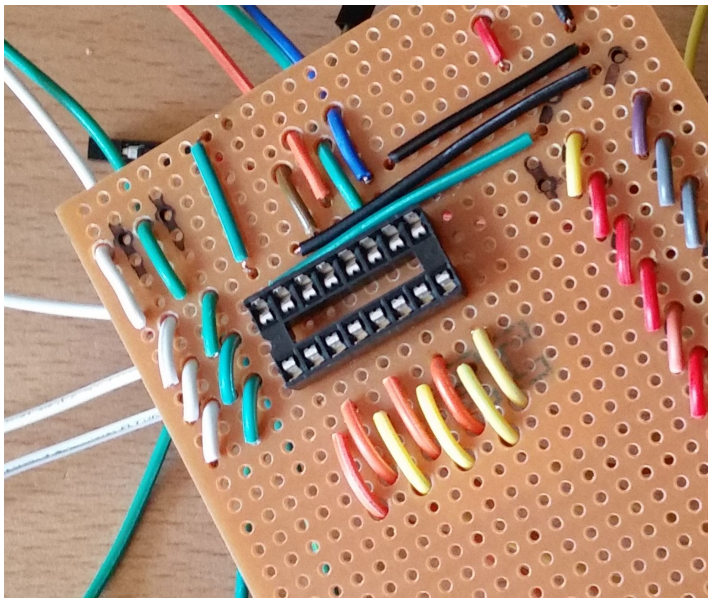


Figure 5.3: The back of the populated stripboard, showing the IC socket, and "flywires" connecting the MCP3008 to the power rails.

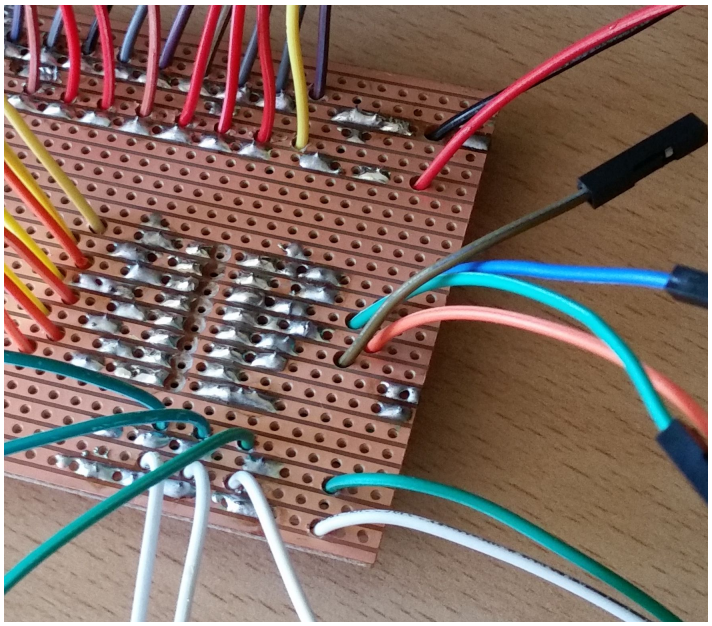


Figure 5.4: The soldered side of the populated stripboard, showing the strip isolation between the MCP3008 pins, solder joins, and wire routing holes.

Jumper cables were then soldered to appropriate strips on the board to provide various connections. This allowed for the MCP3008 to be connected to the Raspberry Pi's GPIO, and to create 3.3V and ground power rails on the stripboard, by connecting them to the appropriate GPIO power pins (figure 5.4).

Similarly, two more rails were created for the LFO and envelope output pins in the same manner, as these required 3 ports on the patchbay, but as they share the same value at all times, they can be connected to the same GPIO pin.

Additionally, these jumper sockets were also connected to the components, such as the 3.5mm patchbay ports, potentiometers, and the push-to-make switches. This allowed the ports and switches to be directly connected to the appropriate GPIO pins, and allows all of the components to be easily removed and potentially replaced in the case of a fault. Also, second pin on each switch was connected to the ground rail.

In order to connect the MCP3008's power and comparison pins (such as *V REF* and *D GND*) to the power rails, wires were simply soldered between the pins' strips and the corresponding power rails (figure 5.3).

Where jumper cables were soldered to components, the solder joins were wrapped with electrical tape, to provide insulation, preventing potential circuit shorting. The wires and MCP3008 on back of the board were also covered with electrical tape, in order to secure them, and prevent potential shorting with the Raspberry Pi's conductive components (figure 5.5).

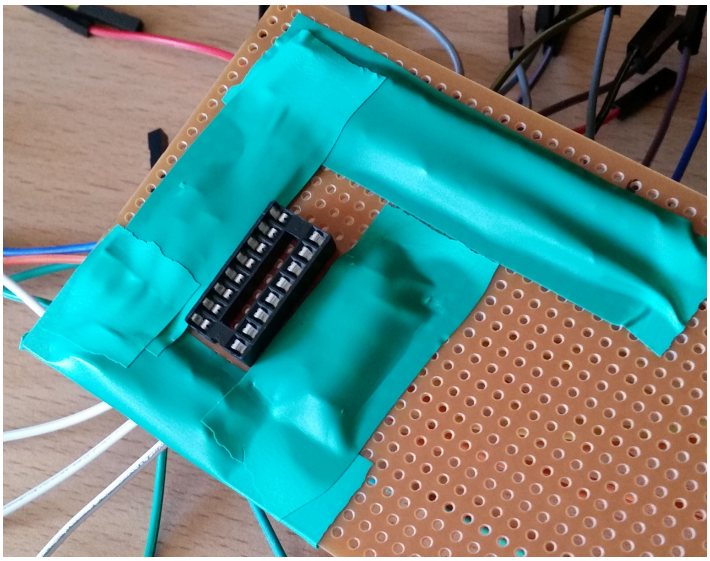


Figure 5.5: The back of the populated stripboard, with connections covered in tape for insulation and security.

5.3 Software Integration

Whilst the libraries used in the implementation were compatible with the Raspberry Pi, there were several issues which were overcome when developing software on the Raspberry Pi. In addition, several steps had to be made in order for system to perform as well as possible, and to provide a simple end-user experience.

5.3.1 OS Installation

First, an operating system must be installed onto an SD card, in order for the Raspberry Pi to function. Raspbian, the Raspberry Pi Foundation's official Debian-based GNU/Linux distribution, was chosen, as it is well documented and widely used, so finding solutions to potential issues would be straightforward. In addition, many of the required packages, such as Python 2.7 are pre-installed on this distribution, so they don't need to be installed afterwards. [67]

Specifically, the "LITE" version of Raspbian was downloaded, as it does not include a desktop environment, which would unnecessarily use system memory and processor cycles, which should be saved for computationally-expensive audio processing instead.

The SD card which the disk image would be written to was inserted into a computer running a GNU/Linux distribution, and unmounted from the filesystem. The "dd" utility was then used to write the image to the card. [68]

Once complete, the SD card was removed from the computer, and inserted into the Raspberry Pi. Once powering the Raspberry Pi on, an automatic utility expands its filesystem to occupy the entire card's capacity.

5.3.2 Configuration

Raspbian includes a configuration utility which allows the user to customise various aspects of the operating system, which is executed by using the command 'sudo raspi-config'. Whilst the default settings are appropriate for most users, several modifications must be made for the system to be appropriate for use as a synthesiser.

Console Boot

As the synthesiser does not require a GUI (graphical user interface), the Raspberry Pi is set to boot in CLI (command line interface) mode in the 'Boot Options' section of the configuration tool. As the LITE version of Raspbian is used, this is set as the default.

However, the boot mode should be changed to 'Console Autologin', in order to skip the authentication process when the system is booted, allowing the synthesiser to be used as soon as the necessary software is loaded, without any user input.

There is little risk in skipping the login process, as no particularly sensitive data is stored on the Raspberry Pi.

3.5mm Audio Output

The Raspberry Pi can output audio from either its HDMI or 3.5mm ports, and the output the device use is normally automatically selected.

However, to prevent potential issues, 3.5mm output is forced, as the synthesiser will use it to output the generated audio to external speakers or headphones. This is set in the 'Advanced Options' section of the configuration tool.

RAM Allocation

As the Raspberry Pi will be running as a "headless" device (no video output or keyboard input, accessed via a network CLI instead [69]), there is no need for a graphical interface, so the GPU (graphics processing unit) does not need to be used.

In order to allow the rest of the system to utilise more memory, the portion of RAM (random access memory) reserved for the GPU can therefore be reduced to the minimum (16MB), via the 'Memory Split' option in 'Advanced Options'. This additional free memory removes the potential memory bottleneck for the rest of the system, particularly the audio processing software.

Remote Access

Whilst implementing software on a Raspberry Pi can be performed on the device itself, connected to a monitor and a keyboard, it is often easier to access the Raspberry Pi over a network via SSH. This allows any other computer running an SSH client to operate the Raspberry Pi in a command line environment. [70]

In order to simplify access to the device, a custom hostname should be set, which is available as an option in 'raspi-config'. For this synthesiser, the hostname "piromancer" was used to match the 'branding' on the top panel of the case.

As the device will be accessible via a network, and therefore the Internet, if the necessary ports are forwarded to the device from the network's router, a new password should be set.

If the default password ("raspberry") is used, there is a security risk, as an automated script run by an attacker can simply attempt to login to any open SSH port on the Internet with the default credentials. This would give the attacker root access to the device, allowing it to be used for potentially nefarious purposes. [71]

A new password can be set for the default user ("pi") via the 'Change User Password' option in 'raspi-config'. Alternatively, the 'passwd' command can be used whilst the user is logged in.

The SSH service should now be enabled in the 'Interfacing Options' section of 'raspi-config'.

The device can now be shutdown using 'sudo shutdown -t 0'. The HDMI video connection can now be removed from the device, and an Ethernet cable can be attached to connect the device to the network.

Once the system has booted by reattaching the power cable, it can be accessed by running the command 'ssh pi@piromancer' on a UNIX-based computer, or by using PuTTY [70] on Windows.

5.3.3 Updating

To ensure that there are no issues with installing future packages, the current packages on the system should be updated.

First, the package list is updated by using 'sudo apt-get update', which ensures that the packages available to be installed or updated are the latest versions.

Next, the currently installed packages are updated by running 'sudo apt-get upgrade'. Alternatively, 'sudo apt-get dist-upgrade' can be used, which fully updates installed packages, whilst considering changes in packages which they depend on. [72]

5.3.4 Jack Audio Server

Pyo can use Jack as an audio server, to interface with the system's sound driver. [73]

The required version of Jack (1.9.10) was installed by running the command 'sudo apt-get install jackd2', after updating the package list. Support for the ALSA audio driver was also installed, using the command 'sudo apt-get install alsaplayer-jack'.

As the Raspberry Pi has relatively limited processing power, several optimisations can be made in the Jack server's configuration to reduce audio output latency, whilst ensuring that stuttering does not occur due to insufficient time to calculate the next audio sample. [74]

The system's Jack server is launched with the command 'jackd -r -p 8 -t 2000 -d alsa –device hw:CARD=0,DEV=0 -S -P -o 2 -n 3 -p 1024 -r 44100 -X seq'. [74]

The *-r* option disables realtime mode, as it is more expensive to compute, and it is not required to achieve sufficient latency. [74]

Using *-p 8* limits the maximum number of ports to 8, removing the cost of using unneeded ports. *-d alsa* sets the audio driver to ALSA, and starts the ALSA-specific options. The *–device* option sets the audio device to use. [74]

Setting *-S* uses "short" 16-bit samples, *-P* uses playback ports only (as audio input is not required), *-o 2* sets only 2 output channels (removing the cost of unneeded channels), and *-X seq* bridges ALSA MIDI to Jack. [74]

The other options, *-n 3*, *-p 1024*, and *-r 44100* set the number of playback latency periods, period length, and sample rate, respectively. These values impact the latency to output audio, which should be as low as possible, to allow the user's inputs to have an almost immediate effect. However, lower latency requires more processing power to calculate samples on time. Therefore, these options were selected to provide minimum latency, without stuttering audio. [74]

The Jack server also requires ownership of the output audio device service, which is set by adding the following to */etc/dbus-1/system.conf*: [75]

```
<policy user="root">
    <allow own="org.freedesktop.ReserveDevice1.Audio0"/>
</policy>
```

As the root user will be executing the Jack server upon boot, these permissions are assigned to root.

Jack requires that it should be run inside an X11 desktop interface session, as X11 includes dbus, which allows applications to interface with one another. [76] However, a workaround is available which allows Jack to run headlessly, by configuring the dbus environment manually. This is achieved by setting an environment variable, *DBUS_SESSION_BUS_ADDRESS*, with the value of *unix:path=/run/dbus/system_bus_socket*. [75]

5.3.5 Patching Integration

In order to simulate patching, the class *patcher.py* was written to track how modules are connected together.

The GPIO pins used for each patchbay port were defined as constants, and variables to track patch connections were defined. These GPIO pins are initialised as their appropriate types, inputs and outputs, and all of the output pins are set as low when the class is instantiated.

The signal path is represented as a queue, so that when the next module in the signal path is detected, it can be appended to the end of the queue. When the first module in the queue needs to be read, it can be popped off the front of the queue, providing its value, and moving the next module to the front.

The auxiliary connections from the LFO and envelope to other modules are simply defined as a list, where new connections can simply be added, and their presence in the list can be

checked when necessary.

The recursive algorithm proposed in the previous chapter was successfully implemented, and is called as the class' *update()* function. This algorithm updates the tracking variables when patches are detected, which can be accessed by *pyohost.py*, as *getModuleQueue()* and *getAuxConnections()*. When the pin values are modified, the program waits for a very short period of time (0.01 seconds) to ensure that the values of the pins are updated correctly, and the correct value is therefore read by the input pins.

5.3.6 MCP3008 Integration

The MCP3008 driver is partly based Limor Fried's example program [66], modified to allow for more than one pin to be read, and to remove unnecessary functionality (i.e. changing the Raspberry Pi's volume).

As the example program was released into the public domain, this modified program will also released in the same manner, as another license would be incompatible.

The GPIO pins used to access the integrated circuit are defined as constants (for future reference in the program), alongside a tolerance value. Also, the "last read" values of each of the MCP3008's pins are initialised as zero in an array.

The *readadc()* function reads the value of the ADC pin passed in as a parameter (referenced by the pin number, from 0 to 7). This value is compared to the pin's last read value that is stored in the previously initialised array. If the difference between these values is greater than the tolerance value, the last read value is updated with the new value, which is returned. However, if the difference is smaller than the tolerance, the unmodified last read value is returned. This process prevents fluctuations in the input voltage from having any impact.

The output value ranges from 0 to 1024, depending on the voltage read by the MCP3008 analogue pin and reference voltage inputs.

The tolerance value (of 10), was determined by testing the smallest value possible where unchanged potentiometers had no fluctuations in their output readings.

Additionally, the *readadc()* function has a "reversed" parameter. If this parameter is set to true when the function is called by *pyohost.py*, the final output value is subtracted from 1024, effectively inverting it, before being returned.

5.3.7 Button Integration

The *buttons* class is used to handle button presses. Two variables to track the properties controlled by the button presses (the oscillator waveform type, and the filter type) are initialised. These variables are integers, with each integer value representing a waveform type or filter type, respectively. The respective types of these properties are stored as constants, with their identifiers, for access by the main *pyohost.py* script.

Additionally, two GPIO inputs for the buttons are initialised. These GPIO inputs were set to use the GPIO's pull up resistor, so that when the button is pressed, and a connection between the GPIO and ground is made, the GPIO reads a high (on) value. [65] [77] [78]

When the class' *update()* function is called, both of the GPIO pins are checked. If a pin reads a high value, the variable which its button controls is incremented. If this value of the variable is greater than the number of its types, the value is set to 1, allowing for these controlled types to be cycled. Additionally, if the button is detected as being pressed, the program waits for a short period of time (0.6 seconds), to prevent a single button press from changing the type multiple times, as well as providing enough time to hear the effect of the new waveform or filter type on the sound produced before cycling to the next.

Therefore, if a button is held down whilst *update()* is called, the appropriate value is incremented, and reset if it becomes greater than its maximum possible value.

The *getWaveFilterTypes()* function returns a tuple of the two tracking variables, allowing the main *pyohost.py* script to access their values.

5.3.8 Pyo Server Development

Pyo Installation

Before the Pyo audio processing library could be used on the Raspberry Pi, it needed to be installed. First, the library's compilation dependencies were installed by executing the command 'sudo apt-get install python-dev libjack-jackd2-dev libportmidi-dev portaudio19-dev liblo-dev libsndfile-dev python-dev python-tk python-imaging-tk python-wxgtk2.8'. [32] The latest version of Pyo's source code was then retrieved by running the command 'git clone https://github.com/belangeo/pyo.git'. The directory was then entered, and Pyo was compiled and installed with 'sudo python setup.py install –install-layout=deb –use-jack'. This also compiled Pyo with support for the Jack audio server as input and output. [73]

Initialisation

The main Python file, *pyohost.py* handles audio processing, and control logic.

Upon execution, this program first loads the other classes, allowing their functions to be accessed for future use. Next, the a Pyo audio server is initialised, allowing for Pyo library's audio processing functionality to be utilised.

A MIDI device is selected, using *s.setMidiInputDevice(3)*. This device is accessed with *midi = Notein(poly=1, scale=1)*, which takes a single MIDI note input stream, of one note at a time, scaling the input keys to the appropriate frequency, in Hertz (Hz). This frequency is stored globally, with *frq = midi['pitch']*.

As only one audio channel is used, Jack is configured to output on both audio output channels, with *os.system("jack_connect pyo:output_1 system:playback_2")*, allowing for both of the speakers on a conventional "stereo" audio system to be used, with the same monaural audio signal emitted from each.

Several variables are initialised to store values from each of the control classes. The envelope's control values read by the MCP3008 are stored in the *envValues* array, and the LFO's control values are stored in the *lfoValues* array. The signal patch queue is stored as *sigConns*, and the auxiliary connection list (connections from the LFO and envelope to other

modules) is stored in *auxConns*. The ID numbers of the waveform and filter types controlled by buttons are stored in a tuple, *btnVals*.

Envelope Module

The envelope is defined as a Pyo *MidiAdsr* envelope, which is triggered upon a MIDI event, with a sustain amplitude based on the input velocity, *midi['velocity']*. The MIDI velocity is a measure of how hard a MIDI key was pressed. When the MIDI velocity is 0, the envelope is released. This envelope's attack, decay, sustain, and release parameters are set as those of the MCP3008's registered envelope controls, tracked by the *envValues* variable.

The envelope is updated with these parameters in the *update_env()* function. The attack, sustain, and release parameters are times (from 0 to 1 seconds), and the sustain parameter controls amplitude of the sustain period, as a percentage of the MIDI velocity, between 0 and 100%. [79]

LFO Module

The LFO module is simulated with the *m_lfo()* function, which returns a Sine wave signal, with the frequency and amplitude defined by the *lfoValues* variable, which is updated with the MCP3008 LFO control values. In order to prevent potential issues with a frequency of 0, 0.1 is added to the LFO's frequency. The frequency and amplitude are also scaled with an input parameter for each, so that they can be modified when used in different modules where necessary. For example, the frequency is generally multiplied by 10, so the LFO's frequency can be set between 0 and 10 Hz.

Update Loop

Pyo's examples [57] generally call a function to display a GUI, which controls Pyo's server, and keeps the signal processing active. However, since this system will be running headlessly, a GUI cannot be used. Instead, a main control loop is called after Pyo is initialised, which ensures the process remains active.

This loop also controls the logic to handle the effects of patching, analogue controls, and button presses. In order to detect changes in these inputs, the variables tracking their values are copied, before being updated. If the updated variable is not identical to the copied variable, then there has been a change.

When a change is detected, the envelope is updated with *update_env()*, and the function *createSignal()* is called, which constructs a new audio signal by recursively checking the signal connection queue, passing through a modified audio signal at each step, until the amplifier module identifier is reached.

Oscillator Module

To simulate an oscillator module, the function *m_oscillator()* makes a local copy of the frequency of the MIDI input note, to use as the frequency of the generated waveform. This

local frequency is modified by the LFO and envelope modules if their respective oscillator connection identifiers are present in *auxConns*. However, the amplitude of the envelope is divided by its *sustain* property, to ensure that the frequency in the sustain period is the same as that of the note input. The function finally returns the waveform that matches the waveform type identifier that is stored in *btnVals*.

Filter Module

The *m_filter()* function first sets a local cutoff frequency variable to that of the input note, and modifies it in the same manner as in *m_oscillator()* for the LFO and envelope inputs. The function returns a filtered version of the input signal provided as a parameter. The filter used is that of the filter type identifier stored in *btnVals*.

Amplifier Module

An amplifier module is implemented with the *m_lfo()* function, which sets a local *gain* value, and takes an input signal as a parameter. The *gain* is replaced by an LFO input if the appropriate connection identifier is present in *auxConns*. If *auxConns* contains the envelope connection identifier, the *gain* is multiplied by the envelope. The input signal's amplitude is multiplied by the *gain* and returned.

Execution

This program is executed via a shell script (*init.sh*), which stops any existing Jack servers, then starts a Jack server to be used by the Pyo server, before executing the main Python script.

5.4 Testing

In order to ensure that each control system of the synthesiser functions correctly, a program was developed for each, to facilitate manual testing.

These test programs initialise the class which they are testing, before executing a main loop, with the following functionalities:

- *patchtest.py* - calls the update function of the patcher class, and prints the signal and auxiliary connection variables to the terminal
- *mcptest.py* - prints the value of each ADC pin read by the MCP3008 driver class, and repeats the process, but with the *reversed* parameter enabled.
- *buttontest.py* - initialises the GPIO pin of each button, in the same manner as the implementation of the button class, and checks each GPIO pin to detect if its button is pressed, printing a message identifying the button if it is pressed. If neither button is pressed, "Nothing Pressed!" is printed to the terminal instead.

In order to test patching, both possible signal path patch routes (Oscillator to Amplifier, and Oscillator to Filter to Amplifier) were made, and the output of *patchtest.py* was compared to the expected output. This test confirmed that this functioned as intended. The same procedure was performed for each possible auxiliary connection (connections from the LFO and envelope modules to the other modules), using the output of *patchtest.py* to ensure that each correct connection identifier was present in the auxiliary connections list. This test confirmed that these auxiliary connections functioned as intended.

Similarly, each of the MCP3008 analogue potentiometer inputs were tested, by turning each in order, and checking that the corresponding maximum and minimum values reported by *mcptest.py* were close to 0 or 1024, when rotated fully in either direction, with a linear change in the value whilst being turned. Additionally, their *reversed* values should have the opposite behaviour to this. It was deemed acceptable if the value started from 1024 at a fully anti-clockwise position, changing towards 0 at a fully clockwise position, or vice-versa, as some potentiometers could be wired with the positive and ground rail connections as the inverse of others, and this can simply be corrected by setting their *reversed* parameter to true.

The button inputs were tested by running *buttontest.py*, and observing the outputs based upon button presses. If neither button is pressed, "Nothing Pressed!" should be output. If the "waveform" button is pressed, "Osc Waveform!" should be output, and if the "filter type" button is pressed, "Filter Type!" should be printed. If both buttons are pressed, both messages should be printed to the terminal. Conducting this test provided the expected results, showing that the button checking system works as intended.

5.4.1 Performance Issues

Initial manual tests showed that each component functioned properly on the Raspberry Pi's hardware, and the simple Pyo synthesis prototype also ran correctly.

However, when all of these elements were combined during the development of the final synthesiser software, the Raspberry Pi's audio began to stutter. As more elements were implemented, and the complexity of the synthesiser increased, this issue became much more prevalent.

Therefore, the original Raspberry Pi Model B was deemed as inadequate to achieve this project's goals, as this issue would persist without removing considerable functionality from the synthesiser, such as the ability to patch modules together, a key aim of the project.

5.5 Raspberry Pi 3

As the original Raspberry Pi (Model B) provided insufficient performance to run the audio synthesis processing software required to implement the synthesiser, a Raspberry Pi 3 (Model B) was used for the final implementation instead, as it was the most capable Raspberry Pi computer available, offering several advantages over the original Model B.

5.5.1 Performance

In particular, the Raspberry Pi 3 has a much faster processor than the original, with four 1200 MHz cores, compared to one 700MHz processor core. [80] [81]

The updated synthesiser software was executed on the Raspberry Pi 3, using the same configuration as the original, and functioned correctly, with no audio stutter.

The manual tests were also executed to ensure that there was no change in any other functionality. The MCP3008 driver and buttons were both shown to function correctly. However, the patching test highlighted a difference in the GPIO functionality.

5.5.2 GPIO Configuration

The first 26 GPIO pins on the Raspberry Pi 3 are identical to those on the original Model B. Therefore, none of the pin connections had to be changed when using the Raspberry Pi 3 instead. [4] [82]

However, testing the patching functionality with *patchtest.py* showed that the Raspberry Pi 3 sometimes misreported some GPIO pins as reading a high value when a jumper cable was connected, even if this cable wasn't connected to anything.

This issue was resolved by forcing the GPIO inputs to use pull down resistors in *patcher.py*, so that the GPIO is always grounded when there is no input voltage. [78] Therefore, this was possibly caused by the Raspberry Pi 3 using a different default resistor mode than the original, or that there may be a different tolerance for floating pins.

5.5.3 Wi-Fi

Additionally, this model includes an integrated Wi-Fi and Bluetooth radio chip [80], removing the necessity to use Ethernet to connect to the device via SSH.

To configure a new access point, "*/etc/wpa_supplicant/wpa_supplicant.conf*" must be edited, with the following added, with fields replaced with the appropriate credentials:

```
network={
        ssid="Access Point SSID (name)"
        psk="Access Point Password"
        id_str="Local Identifier (for more advanced configuration)"
}
```

After booting, the device automatically connects to an available access point listed in this file.

5.6 Final Assembly

Due to space limitations inside the casing, the synthesiser was assembled by connecting more inaccessible components together before the others.

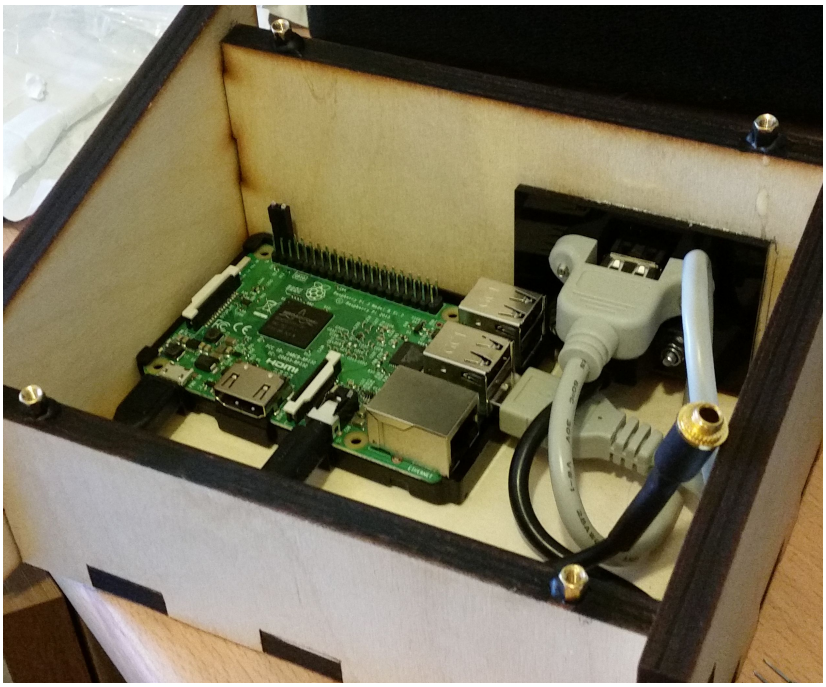


Figure 5.6: The connected panel mount cables.

Initially, the USB panel mounts were bolted into place on the back of the case, and connected to the Raspberry Pi inside the casing (figure 5.6).

The potentiometers, buttons, and 3.5mm patching ports were mounted to the front panel with spacers and their locking nuts (figure 5.7). Knobs were also attached to the potentiometers, in order to provide an indication of how much the potentiometers are turned.

Some of these components were then connected to the stripboard via the jumper cable sockets. Each potentiometer was connected to the 3.3V and ground rails, as well as an MCP3008 analogue input pin. One pin on each switch was connected to the ground rail, and the LFO and envelope output ports were connected to their appropriate rails (figure 5.8).

The stripboard was connected to the appropriate Raspberry Pi GPIO pins via the jumper cables, specifically the power rails, MCP3008 connections, and the LFO and oscillator output rails. These connections were all made on one side of the stripboard, allowing for it to be lifted up for access to the remaining GPIO pins.

The remaining patchbay ports and switch connections were each connected to their corresponding GPIO pins using jumper cables. The Raspberry Pi and stripboard were lifted out of the case to provide easier access to the pins whilst connecting them, before being moved back into place.

Finally, the 3.5mm audio output panel mounting cable was connected to the Raspberry Pi's 3.5mm jack, and attached to front panel, which was then screwed into place using the standoffs attached to the case (figure 5.9).

Figure 5.7: Potentiometers mounted on the front panel.

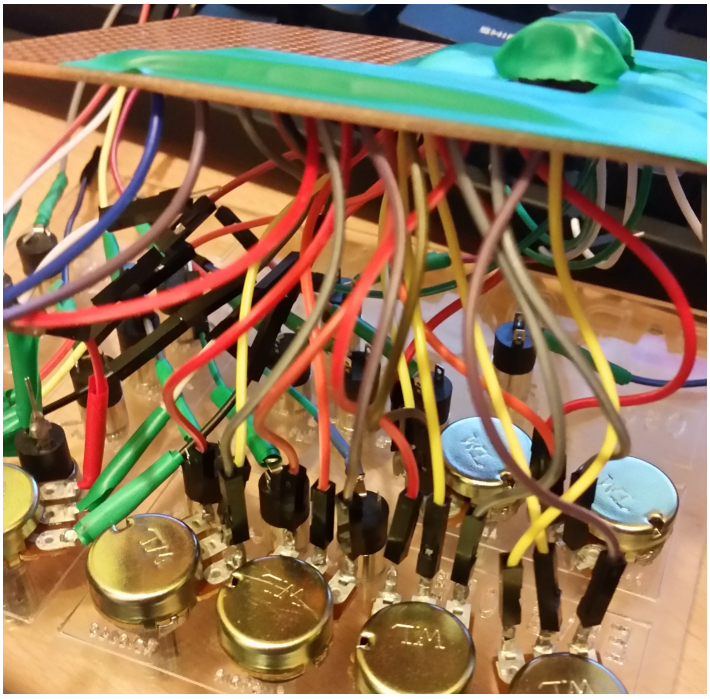


Figure 5.8: Connecting the appropriate front panel components to the stripboard.

Figure 5.9: The fully assembled synthesiser.

5.7 Execute on Startup

The Raspberry Pi was configured to execute the synthesis software on boot, removing the requirement for the user to connect to the system via SSH to start the software before being able to use it.

A script named *startPiromancer.sh* was created in the project's directory (`/home/pi/piromancer-synth`), containing the following:

```
export DBUS_SESSION_BUS_ADDRESS=unix:path=/run/dbus/
        system_bus_socket
cd /home/pi/piromancer-synth/synth
./init.sh
```

This script first exports the environment variable required for Jack to run headlessly. Next, the script navigates to the directory containing the synthesis software, and executes the initialisation script.

Cron [83] is used to execute this script when the Raspberry Pi boots. The command 'sudo crontab -e' is used to edit the system's cron jobs. The line "`@rebootsh /home/pi/piromancer-synth/startPiromancer.sh >/home/pi/logs/cronlog2>&1`" is added to this list, creating a cron job to execute the script (as the root user) when the system boots, and insert the script's output text (including initialisation of the Jack audio server) into a log file, for debugging purposes. [84]

A directory to store these logs is created in the 'pi' user's home directory by executing the command 'mkdir /home/pi/logs'.

5.8 Source Release

The source code and additional materials, including CAD drawings, circuit schematics, the GPIO pinout table, test programs, and the user manual were released in a public GitHub repository.

A brief installation guide was also written for this repository, and the source code was released under the GPLv3 license [53], with the exception of *mcpaccess.py*, which was released into the public domain, in the same manner as the example program it was based upon. [66]

This repository is available at `https://github.com/BenClegg/piromancer-synth`.

Chapter 6

Results and Discussion

6.1 Implementation Challenges

Whilst building the synthesiser and implementing its software, several issues were encountered which required a workaround to overcome.

The 3.5mm panel mount cable used for audio output did not lock a 3.5mm jack into place, causing considerable noise, output through only one of the two audio channels, and the possibility of disconnection by simply moving the cable. To resolve this problem, an audio splitter cable was connected to the Raspberry Pi instead, with one of its outputs connected to the original cable, and the other routed through the gap underneath the patchbay panel. This allowed the existing panel mounted output to be used as a 'monitoring' connection, whilst the other, fully functional output can be connected to the main output device for improved clarity, and no risk of disconnection.

Similarly, the 3.5mm patch ports have the same issue regarding jack locking. However, since there was insufficient time to solder new ports, and the connections still functioned correctly, these ports were still used. The participants in the study were informed of this, to prevent them from exerting excessive pressure on the ports in an attempt to make a patch cable lock into place.

The *buttons.py* class originally implemented button checking as threads, with a thread for each button, which simply incremented the appropriate variable if a button was pressed, and added a wait, in order to prevent signal bouncing. This system worked when tested alone, however, when these threads were utilised in the audio synthesis program, they occupied too much CPU time, as they constantly checked button presses. This caused the audio output to stutter. This issue was resolved by making a single update function perform the same functionality inside the main program's control loop, which repeats around every two seconds. Using this approach required an acceptable compromise that the buttons had to be held down to toggle between waveform or filter types.

After assembling the synthesiser, the two potentiometers used as controls for the amplifier's gain and the filter's cutoff frequency were found to be non-functional. Investigating the issue showed that this was caused by unstable connections between the jumper cables used to

Figure 6.1: The completed synthesiser, as used in the study.

connect the potentiometers to the breadboard, as these potentiometers used cables soldered to them to reach their jumper sockets, whilst the others could be connected directly. This issue highlights a considerable limitation of using sockets to connect components, and would be avoided by directly soldering a wire from the potentiometers to the board instead.

To overcome this issue, the filter's cutoff frequency was set to be that of the MIDI note input, which still allows the filter to alter the timbre of the input signal. The amplifier's gain was set as a constant value, as the envelope and LFO inputs still provided functionality for the amplifier, and the output audio's volume can still be controlled using an inline volume controller from the audio output port. The two potentiometers were removed to prevent potential confusion when participants used the synthesiser.

6.2 Final System

The fully implemented synthesiser (figure 6.1) successfully functions as a software based semi-modular monophonic synthesiser. Modules can be patched together as intended, with the expected outputs.

Whilst updating the audio signal with changed patches, button presses, or parameter changes is noticeable, the recalculation of the signal is a very fast process, allowing for sounds

to be designed for use effectively. Since the patches of modular synthesisers are rarely altered during a performance, this recalculation of an audio signal is acceptable.

Publicly releasing the project's source code with a permissive software licence fulfils the aim of developing the synthesiser as free software. This also allows the software to be easily used as a potential teaching resource.

The synthesiser's casing protects its sensitive electrical internals, since they cannot receive an impact (such as being dropped) directly. This casing is also small enough to remain very portable, at approximately 17cm wide, 14cm deep, and 9.5cm high. This was particularly beneficial when performing the study, as the synthesiser was easily transported to each of the participants, since it fit comfortably inside a small backpack, alongside the necessary MIDI keyboard, power supply, and patch cables.

6.3 Limitations

As this project replicates a traditional monophonic semi-modular synthesiser, only one note's frequency can be output from the oscillator at once, preventing chords from being played. However, Pyo is capable of taking multiple simultaneous MIDI note inputs, and outputting several signals of the appropriate frequencies, each with their own envelope. This project could be adapted into a more contemporary polyphonic semi-modular synthesiser by simply changing this setting, and by replacing the filter's cutoff frequency with an analogue input, since only one frequency can be used in the filter's current configuration.

When no audio is generated, such as when an envelope is used, but no MIDI key is pressed, there is noise emitted from the audio output. This may be due to electrical interference in the Raspberry Pi, or due to the Jack audio server. This issue could prevent the synthesiser from being used in some professional audio applications.

Playing different notes in rapid succession can cause some of the key presses to not register, preventing the sound output from changing accordingly. This is due to the MIDI controller reporting that the first key pressed is still being held down, as it has not reset to its idle position, which it must be at before another key can be read by the Pyo server. However, a MIDI sequencer which can force a note's velocity as zero would not encounter this issue.

6.4 Study Results

6.4.1 Method

The volunteering participants were provided "manual" to refer to whilst using the synthesiser, which documents the functions of each synthesiser, and includes a simple tutorial which explains how to patch various modules together to generate a specific sound, in order to familiarise them with the patching system and the controls of each module.

After their session using the synthesiser, the participants completed an anonymous survey, in which they were asked to rate their opinions on various statements using a scale from 1 to 7, where 1 is "strongly disagree", 7 is "strongly agree", and 4 is a neutral response.

Whilst the other questions relate directly to the synthesiser, the survey first asks the participants to state their musical experience, including their experience with electronic music and modular synthesisers. These responses will be used to infer an explanation for more divisive responses in the rest of the survey.

Finally, the participants were given the opportunity to submit an optional written response, to suggest potential improvements, to highlight the synthesiser's strengths and weaknesses, and to provide additional comments.

6.4.2 Survey Analysis

Clarity of Design

In general, the participants agreed that "it was clear how modules could be patched together", with an average response of ~ 5.6.

However, the two minimum responses were 4, indicating a neutral opinion. Therefore, the patching interface could be improved, perhaps by adding colours for each module connection, or by making the labels larger to improve readability.

Impression of Control

Whilst the users tended to feel that their "actions on the synthesiser directly influenced the sound generated" (with an average response of ~ 5.9), one participant gave a neutral response.

Since this participant indicated that they have considerable experience with electronic music performance or production, this may be attributed to the fact that the synthesis software must regenerate an audio signal when a parameter is changed, compared to the direct control provided by an analogue synthesiser, which this participant could have prior experience with.

Layout Organisation

The participants all agreed that "the layout of the modules made it easy to patch them together, and access their (controls)", with an average response of ~ 6.3, a minimum of 5, and a maximum of 7.

Manipulation from the "Zero State"

Whilst most participants were shown to agree that they "could easily start to patch modules together to design a sound, when no patches were originally connected" (with an average response of ~ 5.6, and a maximum of 7), some users stated that they found this to be complex, shown by the minimum response of 3.

However, in this participant's additional comments, they state that they "could have used more time to learn what my actions were doing when testing, but had a limited time frame to test." This participant may have been able to learn how to patch modules together from

the "zero state" if they had more time to use the synthesiser. This could be regarded as a limitation, however, as there may be a more effective method to teach a user how to operate the synthesiser, compared to a brief tutorial and a user manual.

Visual Appeal

To varying extents, each of the participants stated that they "like the aesthetic design of the synthesiser", with an average response of ~ 6.4, and a minimum of 5.

Synthesis Capabilities

The responses indicated that the participants were generally impressed by "the sounds that could be produced with the synthesiser", with an average response of ~ 5.7. However, one participant with past experience with other synthesisers gave a neutral response of 4.

This response may be due to their past experience, as some other synthesisers are capable of producing a wider range of more complex sounds, with little interference, compared to this simpler synthesiser implementation.

Simplicity

The statement "I understood how to patch modules together to make a particular sound" received a varied response, with an average of ~ 5.0, a maximum of 7, and a minimum of 3.

Since the minimum response disagrees with the statement to an extent, and this user stated that it was clear how the modules can be patched together, perhaps the user manual could be improved, to be more concise, and offer more guides for designing sounds.

Utility as a Musical Instrument

The participants gave very mixed responses to the statement "I would like to use the synthesiser in future performances, production, or jam sessions", with an average response of ~ 4.7, a maximum of 7, and a minimum of 2.

This mixed response may be due to the range of applications proposed, as the synthesiser's limitations (such as the noise produced where silence should be present) would make it unsuitable for professional performances, yet wouldn't have an impact on music production, since this noise can be removed from a recording of the synthesiser's output.

Therefore, each participant may have considered a different use case when responding to this statement.

Interest in Source Code Release

The statement "I would be interested in using publicly released source code of the synthesiser to either modify it, or create my own" also received mixed responses, with an average of ~ 4.7, a minimum of 2, and a maximum of 7.

This diverse results set is probably due to the difference in interests of the participants. Some participants may have a strong interest in electronics and programming, whereas others may not. There were no neutral responses, and the positive responses were strong (either 6 or 7), indicating that those who are generally interested in programming or electronics would be interested in access to the source code, under a free software licence.

Suitability as an Educational Resource

All of the participants agreed that "the synthesiser would be an effective teaching tool for programming, music, or both", to varying extents, with an average response of "6.1", and a minimum of 5.

Additional Comments

One participant with past electronic music experience stated that it was "very hard to tell the different wave forms apart at the volume level", and that the "background hiss did not help." This comment highlights the limitation of the lack of gain control, and the electrical noise issues. These issues could be overcome by using a more advanced audio interface, and by repairing the amplifier's gain potentiometer in the future, given enough resources.

Another participant commented that they "have no experience with synthesisers and my only musical experience is playing the guitar at an extremely low level, but it was simple to use and I found it very fun to do so. I'd imagine if I was interested in making music, especially through the use of synthesisers, this would be an extremely useful tool and a great place to start." This reinforces the other survey results which indicate that the synthesiser would be an effective resource for music (and programming) education.

Another user remarked that the synthesiser is "a fantastic device. I love (how) easy it is to use, and to make great sounding pieces of music with! I want to know more about the different ways to connect it up, if there are any." This statement shows that it would be beneficial to provide multiple methods of interfacing with the synthesiser, besides a MIDI keyboard.

6.5 Future Research

This project shows the potential of using the Raspberry Pi for audio synthesis and processing, which could be explored further in several manners:

- Modifying the synthesiser to support polyphony;
- Implementation of additional modules, such as a chorus, a ring modulator, or a sampler;
- Creating other synthesisers which employ different synthesis methods, such as Frequency Modulation;

- Utilisation of alternative MIDI input devices, such as a sequencer built using a Raspberry Pi;
- Using Pyo on a Raspberry Pi as an audio processor, such as a "talkbox" style vocoder.

Chapter 7

Conclusions

This project investigated the process of creating a modular synthesiser using a Raspberry Pi, which should be appropriate for use as an educational tool, and as a musical instrument.

The project also aimed to produce freely available resources of a high quality, for the use of individuals interested in developing a similar system.

In order to find a suitable approach to fulfil these goals, research was conducted on various synthesis techniques, modular synthesisers, audio processing, musical interfaces (such as MIDI), relevant hardware, appropriate audio processing software, and manufacturing processes.

This research revealed that creating a traditional modular synthesiser would unfeasible, due to the costs of multiple Raspberry Pis and audio interfaces, and that implementing a semi-modular system using the Pyo audio processing library would be an appropriate compromise.

The research process also highlighted that the Raspberry Pi's GPIO pins are inappropriate for audio transmission in a similar manner to a conventional semi-modular synthesiser, and should instead be used to simulate signal patching by determining the path of an audio signal, and using this information to process the audio appropriately.

Several materials to be used in the manufacture of the synthesiser were designed, including CAD drawings for the synthesiser's casing, and electronic circuit schematics showing how an MCP3008 Analogue to Digital Converter chip could be connected to the Raspberry Pi, allowing for potentiometers to control parameters of different modules.

A casing for the synthesiser was manufactured by using a laser cutter to cut plywood and acrylic panels, using the designed CAD drawings. Stripboard was used to create a circuit to connect the MCP3008 and other components to the the Raspberry Pi. Jumper cables were soldered to the stripboard to provided GPIO and component connections.

Software was developed to implement the audio processing features of LFO, envelope, oscillator, filter, and amplifier modules. Object oriented programming was used to integrate different functionalities, such as buttons to toggle between the oscillator's waveforms, a driver for the MCP3008 ADC chip (based on a public domain example program), and a patch detection system. Recursive algorithms were employed to detect and apply patches.

Several challenges were encountered during this implementation process. For example, loose jumper cable connections to potentiometers prevented them from functioning correctly, revealing that jumper cables are inappropriate for use other than to connect components directly to the Raspberry Pi's GPIO pins.

The project's source code and supporting materials were publicly released via GitHub, under an open software licence, allowing an individual to modify the code, and release their own version. This fulfilled the aim of providing freedom over the synthesiser's software to hobbyists.

The completed synthesiser was evaluated via a study, in which participants created sounds using the synthesiser, and answered a survey, providing their opinions on the synthesiser.

This survey showed that the synthesiser has some limitations, such as the output of noise where there should be silence.

However, the survey also shows that the synthesiser fulfils its aims to an extent, with all of the participants agreeing that it would be suitable as an educational resource, and some participants professed their interest in having access to the source code in order to create their own.

Bibliography

[1] Peter Kirn. Moog mother-32 wants to be your intro to modular synthesis. [Online; accessed 20-November-2016] http://cdm.link/2015/10/moog-mother-32-wants-to-be-your-intro-to-modular-synthesis/.

[2] Martin Russ. *Sound Synthesis and Sampling, 2nd Edition.* Focal Press, Linacre House, Jordan Hill, Oxford, OX2 8DP, 2004.

[3] Ian Smith. Moog mother-32 block diagram. [Online; accessed 06-February-2017] http://ian-d-smith.me.uk/electronic-music-notepad/moog-mother-32-block-diagram.

[4] Various. Gpio: Raspberry pi models a and b. [Online; accessed 19-April-2017] https://www.raspberrypi.org/documentation/usage/gpio/.

[5] Moog. Moog mother-32. [Online; accessed 20-November-2016] https://www.moogmusic.com/products/semi-modular/mother-32.

[6] Beau Sievers. Synthesis basics. [Online; accessed 11-November-2016] http://beausievers.com/synth/synthbasics/.

[7] F. Richard Moore. *Elements Of Computer Music.* Pearson Education, University of California, San Diego, 1990.

[8] Scott Rise. Subtractive synthesis. [Online; accessed 16-November-2016] http://synthesizeracademy.com/subtractive-synthesis/.

[9] Scott Rise. Additive synthesis. [Online; accessed 16-November-2016] http://synthesizeracademy.com/additive-synthesis/.

[10] Scott Rise. Fm synthesis. [Online; accessed 16-November-2016] http://synthesizeracademy.com/fm-synthesis/.

[11] Scott Rise. Wavetable synthesis. [Online; accessed 03-December-2016] http://synthesizeracademy.com/wavetable-synthesis/.

[12] Tom Wiltshire. Phase distortion synthesis. [Online; accessed 03-December-2016] http://electricdruid.net/phase-distortion-synthesis/.

[13] Prof. Jeffrey Hass. Introduction to computer music: Volume one - chapter three: Midi. [Online; accessed 13-November-2016] http://www.indiana.edu/~emusic/etext/MIDI/chapter3_MIDI.shtml.

[14] Various Authors. Introduction to osc. [Online; accessed 13-November-2016] http://opensoundcontrol.org/introduction-osc.

[15] Christian Moraga. Aira modular: What is control voltage? [Online; accessed 15-November-2016] https://www.rolandcorp.com.au/blog/aira-control-voltage.

[16] Digital-Growth. Using cv and gate. [Online; accessed 15-November-2016] http://analoguesolutions.com/using-cv-and-gate/.

[17] Pittsburgh Modular Synthesizers LLC. Midi 3 - midi to cv converter. [Online; accessed 13-November-2016] http://pittsburghmodular.com/midi3/.

[18] Dr. David Berners. The inner workings of the moog multimode filter. [Online; accessed 17-November-2016] http://www.uaudio.com/blog/moog-multimode-filter-design/.

[19] Scott Rise. Ring modulator - the synthesizer academy. [Online; accessed 16-November-2016] http://synthesizeracademy.com/ring-modulator/.

[20] Dave Hunter. Effects explained: Modulation iivibrato, tremolo, octave divider, ring modulator. [Online; accessed 16-November-2016] http://archive.gibson.com/en-us/Lifestyle/ProductSpotlight/GearAndInstruments/vibrato-tremolo-octave-divider/.

[21] Doepfer Musikelektronik. A-112 sampler / wavetable module. [Online; accessed 20-November-2016] http://www.doepfer.de/a112.htm.

[22] Vintage Synth Explorer. Akai mpc2000 / mpc2000 xl. [Online; accessed 20-November-2016] http://www.vintagesynth.com/akai/mpc2000.php.

[23] Joonas Pihlajamaa. Benchmarking raspberry pi gpio speed. [Online; accessed 23-November-2016] http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/.

[24] B_E_N SparkFun Community. What is an arduino? [Online; accessed 23-November-2016] https://learn.sparkfun.com/tutorials/what-is-an-arduino.

[25] Paul McWhorter. Raspberry pi lesson 32: Options for analog input on the raspberry pi. [Online; accessed 23-November-2016] http://www.toptechboy.com/raspberry-pi/raspberry-pi-lesson-32-options-for-analog-input-on-the-raspberry-pi/.

[26] Michael Sklar. Analog inputs for raspberry pi using the mcp3008. [Online; accessed 12-April-2017] https://learn.adafruit.com/reading-a-analog-in-and-controlling-audio-volume-with-the-raspberry-pi.

[27] Peter Manning. *Electronic and Computer Music, Revised and Expanded Edition*. Oxford University Press, Inc., 198 Madison Avenue, New York, New York 10016, 2004.

[28] Barry L. Vercoe et. al. The canonical csound reference manual, May 1997. [Online; accessed 14-November-2016] `http://www.csounds.com/manual/html/`.

[29] Barry L. Vercoe and Mike Berry. cpsmidi, May 1997. [Online; accessed 14-November-2016] `http://www.csounds.com/manual/html/cpsmidi.html`.

[30] Olivier Bélanger. Pyo - dedicated python module for digital signal processing. [Online; accessed 15-November-2016] `http://ajaxsoundstudio.com/software/pyo/`.

[31] Olivier Bélanger. pyo - python dsp module - github. [Online; accessed 15-November-2016] `https://github.com/belangeo/pyo`.

[32] Paul Walsh. How to install python pyo on raspberry pi. [Online; accessed 16-November-2016] `https://gist.github.com/pwalsh/8594869`.

[33] SuperCollider Community. Supercollider. [Online; accessed 15-November-2016] `http://supercollider.github.io/`.

[34] Sam Aaron. Supercollider on the raspberry pi. [Online; accessed 15-November-2016] `http://sam.aaron.name/2012/11/02/supercollider-on-pi.html`.

[35] Pure Data Community. Pure data - pd community site. [Online; accessed 15-November-2016] `https://puredata.info/`.

[36] ChucK Team. Chuck. [Online; accessed 15-November-2016] `http://chuck.cs.princeton.edu/`.

[37] Sonic Pi Core Team Sam Aaron. Sonic pi. [Online; accessed 26-November-2016] `http://sonic-pi.net/`.

[38] Raspberry Pi Foundation. Sonic pi lessons. [Online; accessed 26-November-2016] `https://www.raspberrypi.org/learning/sonic-pi-lessons/`.

[39] Teenage Engineering. Pocket operator. [Online; accessed 18-November-2016] `https://teenage.engineering/products/po`.

[40] Kate Cummins. The rise of additive manufacturing. [Online; accessed 21-April-2017] `https://www.theengineer.co.uk/issues/24-may-2010/the-rise-of-additive-manufacturing/`.

[41] Shi-Joon Yoo, Omar Thabit, Eul Kyung Kim, Haruki Ide, Deane Yim, Anreea Dragulescu, Mike Seed, Lars Grosse-Wortmann, and Glen van Arsdell. 3d printing in medicine of congenital heart diseases. *3D Printing in Medicine*, 2(1):3, 2016.

[42] Andreas A. Giannopoulos, Leonid Chepelev, Adnan Sheikh, Aili Wang, Wilfred Dang, Ekin Akyuz, Chris Hong, Nicole Wake, Todd Pietila, Philip B. Dydynski, Dimitrios Mitsouras, and Frank J. Rybicki. 3d printed ventricular septal defect patch: a primer for the 2015 radiological society of north america (rsna) hands-on course in 3d printing. *3D Printing in Medicine*, 1(1):3, 2015.

[43] F. Rengier, A. Mehndiratta, H. von Tengg-Kobligk, C. M. Zechmann, R. Unterhinninghofen, H.-U. Kauczor, and F. L. Giesel. 3d printing based on imaging data: review of medical applications. *International Journal of Computer Assisted Radiology and Surgery*, 5(4):335–341, 2010.

[44] Charles McMillin and John Harry. Laser machining of southern pine. *Forest Products Journal*, 21(10):35–37, 1971.

[45] B. Radu-Eugen, T. Sorin, B. Cristina, and B. Octavian-Constantin. Improving the dynamic behavior and working accuracy of the cnc laser cutting machines. In *2012 12th International Conference on Control Automation Robotics Vision (ICARCV)*, pages 1642–1647, Dec 2012.

[46] C. Liu, Y. Hu, and B. Fu. The state-of-the-art and development tendency of cnc laser cutting machine at home and abroad-an overview. In *2012 Spring Congress on Engineering and Technology*, pages 1–4, May 2012.

[47] M. Stevens, E. Ball, and A. Protogeros. Process compensation and printed circuit board manufacture. *The International Journal of Advanced Manufacturing Technology*, 8(2):85–90, 1993.

[48] gEDA Project. geda project. [Online; accessed 19-April-2017] `http://www.geda-project.org/`.

[49] M. Herbert. Homebuilt: Building a laser ukulele. *IEEE Potentials*, 34(4):6–10, July 2015.

[50] BusBoard Prototype Systems. Stripboard-3u - general purpose stripboard - datasheet. [Online; accessed 22-April-2017] `http://www.mouser.com/ds/2/58/BPS-DAT-(ST3U)-Datasheet-771561.pdf`.

[51] Joshua Porter. Principles of user interface design. [Online; accessed 19-April-2017] `http://bokardo.com/principles-of-user-interface-design/`.

[52] The University of Sheffield. Human participants in research and applying for ethical review. [Online; accessed 19-April-2017] `https://www.sheffield.ac.uk/ris/other/gov-ethics/ethicspolicy`.

[53] Brett Smith. A quick guide to gplv3. [Online; accessed 24-April-2017] `https://www.gnu.org/licenses/quick-guide-gplv3.html`.

[54] Bjrn Arlt. Mono/fury. [Online; accessed 03-December-2016] `https://www.fullbucket.de/music/monofury.html`.

[55] Korg. Mono/poly software synthesizer. [Online; accessed 20-November-2016] `http://www.korg.com/uk/products/software/korg_legacy_collection/page_3.php`.

[56] The Linux Information Project. Proprietary software definition. [Online; accessed 02-May-2017] `http://www.linfo.org/proprietary.html`.

[57] Olivier Bélanger. Pyo 0.8.0 documentation. [Online; accessed 16-November-2016] `http://ajaxsoundstudio.com/pyodoc/`.

[58] Barnaby Walters. A simple pyo synth. [Online; accessed 20-November-2016] `https://waterpigs.co.uk/articles/simple-pyo-synth/`.

[59] Lars Kellogg-Stedman. python-siggen (github). [Online; accessed 04-March-2017] `https://github.com/larsks/python-siggen`.

[60] HardWired. Playpi / fluxpi. [Online; accessed 04-March-2017] `http://pi.hardwiredonline.co.uk/index.php`.

[61] Trimble Incorporated. Sketchup. [Online; accessed 20-February-2017] `https://www.sketchup.com/`.

[62] LibreCAD. Librecad. [Online; accessed 25-February-2017] `http://librecad.org/cms/home.html`.

[63] Autodesk. Autocad. [Online; accessed 25-February-2017] `http://www.autodesk.co.uk/products/autocad/overview`.

[64] Ben Croston and Anonymous. Rpi.gpio module basics. [Online; accessed 20-April-2017] `https://sourceforge.net/p/raspberry-gpio-python/wiki/BasicUsage/`.

[65] Simon Monk. *Raspberry Pi Cookbook*. O'Reilly Media, 2013. Switch example available in sampler [Online; accessed 20-April-2017] `http://razzpisampler.oreilly.com/ch07.html`.

[66] Limor Fried. adafruit_mcp3008.py - raspbery pi analog input with mcp3008. [Online; accessed 20-April-2017] `https://gist.github.com/ladyada/3151375`.

[67] Raspberry Pi Foundation. Raspbian. [Online; accessed 07-April-2017] `https://www.raspberrypi.org/downloads/raspbian/`.

[68] Raspberry Pi Foundation. Installing operating system images on linux. [Online; accessed 07-April-2017] `https://www.raspberrypi.org/documentation/installation/installing-images/linux.md`.

[69] Rick Leander. What is headless linux? [Online; accessed 10-April-2017] http://smallbusiness.chron.com/headless-linux-33715.html.

[70] Simon Tatham et. al. Putty. [Online; accessed 10-April-2017] http://www.chiark.greenend.org.uk/~sgtatham/putty/.

[71] Scott Howard. Hacking iptv, or why default passwords are bad. [Online; accessed 10-April-2017] https://blog.docbert.org/hacking-iptv/.

[72] Jason Gunthorpe and APT team. apt-get - apt package handling utility - manpage. [Online; accessed 10-April-2017] http://manpages.ubuntu.com/manpages/zesty/en/man8/apt-get.8.html.

[73] Olivier Bélanger. Compiling pyo from sources. [Online; accessed 20-November-2016] http://ajaxsoundstudio.com/pyodoc/compiling.html.

[74] Stefan Schwandter, Jack O'Quin, and Alexandre Prokoudine. jackd manpage. [Online; accessed 29-April-2017] http://manpages.ubuntu.com/manpages/zesty/man1/jackd.1.html.

[75] "micahchips". cannot start jackd without x11. [Online; accessed 29-April-2017] https://www.raspberrypi.org/forums/viewtopic.php?t=57025.

[76] Inc. Software in the Public Interest. Package: dbus-x11 - simple interprocess messaging system (x11 deps). [Online; accessed 29-April-2017] https://packages.debian.org/wheezy/dbus-x11.

[77] Aaron "a1ronzo". Pull-up resistors. [Online; accessed 29-April-2017] https://learn.sparkfun.com/tutorials/pull-up-resistors.

[78] Resistor Guide. Pull up resistor / pull down resistor. [Online; accessed 30-April-2017] http://www.resistorguide.com/pull-up-resistor_pull-down-resistor/.

[79] Olivier Bélanger. Midi handling. [Online; accessed 27-April-2017] http://ajaxsoundstudio.com/pyodoc/api/classes/midi.html.

[80] Michael Rundle. Raspberry pi 3 is the first with built-in wi-fi. [Online; accessed 10-April-2017] http://www.wired.co.uk/article/raspberry-pi-three-wifi-bluetooth-release-price-cost.

[81] Raspberry Pi Foundation. Raspberry pi 1 model b. [Online; accessed 30-April-2017] https://www.raspberrypi.org/products/model-b/.

[82] Andrew Scheller, Clive Beale, Helen Lynn, and Ben Nuttall. Gpio: Models a+, b+, raspberry pi 2 b and raspberry pi 3 b. [Online; accessed 30-April-2017] https://www.raspberrypi.org/documentation/usage/gpio-plus-and-raspi2/.

[83] Raspberry Pi Foundation. Scheduling tasks with cron. [Online; accessed 29-April-2017] `https://www.raspberrypi.org/documentation/linux/usage/cron.md`.

[84] Scott Kildall. Raspberry pi: Launch python script on startup. [Online; accessed 29-April-2017] `http://www.instructables.com/id/Raspberry-Pi-Launch-Python-script-on-startup/`.

Appendices

Csound Prototype

```
import csnd6
c = csnd6.Csound()

#Options
c.SetOption("-odac") #Audio out
c.SetOption("-+rtmidi=portmidi") #Use portmidi
c.SetOption("-M1") #Use midi 1


orcInit = """
sr = 44100
ksmps = 32
nchnls = 2
0dbfs  = 1
"""
startInst1 = """
instr 1
    inote cpsmidi

    kbandin init 1
    midicontrolchange 7, kbandin, 0, 1
    kband = (kbandin * 3000)

    printk2 kband
    ;printi2 inote

"""
sin = """
    asig vco 20000, inote, 1, 0, 1
        ares moogladder asig, kband, 0.9
            outs 0.01 * ares, 0.01 * ares
"""

endIn = """
endin
"""

#Score
score = """
```

```
; Table #1, a sine wave.
f 1 0 65536 10 1
f 0 3600
e
"""

#Compile
c.CompileOrc(orcInit + startInst1 + sin + endIn)
#Read score, which is a dummy frequency table
#This allows for the reading of midi
c.ReadScore(score)
c.Start()

#Infinite loop to process sound blocks.
while (c.PerformKsmps() == 0):
    pass

c.Stop()
c.Cleanup()
```

Pyo Prototype

```
from pyo import *

s = Server() #Load server instance
s.setMidiInputDevice(3) #Set midi device
s.boot() #Boot server

midi = Notein(poly=10, scale=1) #scale=1 : pitch in Hz
vel = Port(midi['velocity'], .001, .5) #Note velocity
frq = midi['pitch'] #Note pitch (frequency in Hz)

osc = SuperSaw(freq=frq, mul=vel) #SuperSaw wave

#Moog Resonant Lowpass scaled to input note
filt = MoogLP(osc, freq=(frq), res=1).out()
a = osc.out()

sc = Scope(a) #Waveform display
sp = Spectrum(a) #Spectrum display

s.gui(locals()) #Display gui
```