UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Designing the Audio Generation Core of a Hardware Synthesiser

Marco Rademan
21561273

Report submitted in partial fulfilment of the requirements of the module
Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at Stellenbosch University.

Supervisor: Prof J. Versfeld

November 2021

Acknowledgements

I would like to thank my dog, Muffin. I also would like to thank the inventor of the incubator; without him/her, I would not be here. Finally, I would like to thank Dr Herman Kamper for this amazing report template.

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
   Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.
2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
   I agree that plagiarism is a punishable offence because it constitutes theft.
3. Ek verstaan ook dat direkte vertalings plagiaat is.
   I also understand that direct translations are plagiarism.
4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
   Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism
5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

21561273 Studentenommer / *Student number*	[illegible] Handtekening / *Signature*
MW Rademan Voorletters en van / *Initials and surname*	**November 2021** Datum / *Date*

Abstract

English

The English abstract.

Afrikaans

Die Afrikaanse uittreksel.

Contents

List of Figures

List of Tables

Listings

Nomenclature

Variables and functions

$y[n]$	A discreet-time signal with samples indexed by variable n.
$x_1[n] * x_2[n]$	The convolution of discreet-time functions.
$h[n]$	The impulse response of a discrete-time system.
$X(f)$	The discreet-time Fourier transform (DTFT) of a function.
$\lfloor x \rfloor$	The floor of a variable x, corresponding to the integer component of x.
$\{x\}$	The fractional part of x.
$\{n_1, n_2, ..., n_K\}$	A set of numbers, not to be confused with the fractional function.
η	The frequency scaling factor in wavetable sampling.
$y[n]_{\uparrow L}$	Upsampling a discreet-time signal by a factor L.
$y[n]_{\downarrow M}$	Downsampling a discreet-time signal by a factor M.
$\text{tri}[\frac{n}{L}]$	A discreet triangular pulse beginning at $-L$ and ending at L , with an amplitude of 1.
$\text{rect}[\frac{n}{L}]$	A discreet rectangular pulse beginning at $\frac{-L}{2}$ and ending at $\frac{L}{2}$, with an amplitude of 1.
$\mathbb{N}$	The set of all natural numbers ($\{1, 2, 3, 4, ...\}$).
$\mathbb{N}_0$	The set of all natural numbers including 0.
$\mathbb{R}$	The set of all real numbers.
$\mathbb{Z}$	The set of all integers ($\{..., -2, -1, 0, 1, 2, ...\}$).
$\text{lerp}(x_1, x_2, \delta)$	Linear interpolation between points x_1 and x_2, by a distance factor δ.

Acronyms and abbreviations

VCA	Voltage-controlled amplifier
ADSR	Attack, Decay, Sustain, Release
VCF	Voltage-controller filter
VST	Virtual studio technology
DAW	Digital audio workstation
FM	Frequency modulation
LUT	Lookup table
LFO	Low frequency oscillator
IR	Impulse response
VCO	Voltage-controlled oscillator
CV	Control voltage
LPF	Low-pass filter
HPF	High-pass filter
BPF	Band-pass filter
FIR	Finite impulse response
IIR	Infinite impulse response
MIDI	Musical instrument digital interface
SMF	Standard MIDI file
SQNR	Signal to quantisation noise ratio
PCM	Pulse-code modulation
DPCM	Differential pulse-code modulation
UART	Universal asynchronous receiver/transmitter
SAI	Serial audio interface
DSP	Digital signal processing
LP12	12 dB/octave LPF
HP12	12 dB/octave HPF
BP12	12 dB/octave BPF
LP24	24 dB/octave LPF
HP24	24 dB/octave HPF
MCU	Microcontroller unit
MPU	Microprocessing unit
IC	Integrated circuit
FPU	Floating-point unit
STFT	Short-time Fourier transform

Chapter 1
Introduction

1.1. Background and motivation

Hardware synthesisers play an important role in all of music today, and has seen a cast expanse in a variety of product lines across the globe, since the inception of analogue sound synthesisers in 1928. A hardware synthesiser refers to a physical system that generates audio signals, using external control inputs that can either be pre-programmed, such as a drum-machine, or manually sent by the user, such as a piano keyboard via the MIDI protocol. These audio systems can be digital, where sound is generated by a MCU, or analogue, generated by the manipulation of VCOs, using a variety of control signals. A few user-specified parameters can often be enough to generate a wide array of interesting and complicated sounds, some of which are iconic and recognisable in old and modern music alike.

Due to the highly competitive nature of the market and complexity of hardware audio generation systems, much of the design and techniques used in products are often proprietary. Low-level design for MCUs, such as the STM32 Cortex M4 and M7 series, are often not well-documented or analysed. Open-source code [5] for these processors are often available for DIY projects, but are usually minimal and leave many of the frequency domain effects and MCU optimisations unexplored. This thesis aims in designing the software for such a system, in a well-documented and analysed fashion, that can be used as a building block for a wide array of more complicated products.

1.2. Problem statement

Wavetable-based audio generation software must be designed. The system must be able to play up to a fixed number of notes, each with an arbitrary frequency, by using on/off note triggers. This makes the system compatible with must forms of user note inputs, whether it be a button or MIDI messages. This system must be designed conceptually, and implemented in C, such that it considers MCU implementation. The system is especially targeted for ARM-based MCUs, such as the STM32 M4 and M7 series.

The system must produce high-quality stereo audio, while taking speed and memory into account. The system must include all the basic synthesis features: volume modulation; filtering and cut-off modulation; ADSR envelope control signals; FM; waveshaping. When all the basic synthesis techniques are designed, modelled and tested, this thesis must provide the foundation for implementation on MCUs, which can also be altered to produce a more complex product. Thus, the system must be designed in a way that allows for easy implementation on hardware, while being input-type and hardware agnostic.

1.3. System specification

1.3.1. Scope

The scope of this thesis is restricted to synthesis software only. This excludes the design and/or implementation of any hardware-related code, such as SAI drivers or MIDI decoders. Anything that cannot be tested and analysed in a entirely software-based workflow is out of scope. Thus, the audio data that is generated by the system must be stored in a file and analysed externally, using a platform such as MATLAB. The following items are within the scope of this thesis:

1. The audio-synthesis core software, that is triggered by note-on and note-off information.
2. The emulation of MIDI input by interpreting SMFs, for testing.
3. User-parameters that can be changed under test conditions to test system functionality.

Even though no hardware is designed, the hardware must be considered when designing the software, to ensure good support for a variety of topologies. Computational efficiency is emphasised in the design, but timing will not be tested, since it is hardware dependent (RISC versus CISC, clock frequency, available peripherals, etc.).

1.3.2. Functional specification

The following items specify the audio generation capabilities, which has been formulated with reference to figure B.1 in appendix B.2, along with including some of the functionality of features provided by other eurorack modules:

1. The user can select the basic waveforms (sine, triangle, sawtooth and square) and interpolate between them, as is common in many wavetable synthesisers (see figure A.2 in the appendix).
2. The user can change the stereo width of the audio by detuning 2 additional oscillators per note and setting stereo width by panning and changing the volume of the additional oscillators.
3. An ADSR envelope, with user set parameters, must control the volume envelope of a single note over time.
4. The user can choose from a selection of filters: HP12, HP24, LP12, LP24 and BP12. The filters can have a user-defined Q, if applicable.
5. The user can set the filter cut-off frequency relative to the fundamental frequency of a note. This is to ensure that all notes have the same timbre at different frequencies. An ADSR envelope, with user set parameters, must control the cut-off frequency off the note over time.
6. The user can perform stereo wave-shaping on the audio, where they can specify a gain value into a waveshaper function. The waveshaper functions available must be the hyperbolic tangent and a sinusoid.
7. The user must be be able to apply sinusoidal vibrato at a specified rate and amount.
8. The instrument must be polyphonic, i.e. can play multiple notes at once.

1.3.3. Technical requirements

The functional requirements stipulated in subsection 1.3.2 can be translated into technical requirements for the software audio core, which can be measured and designed. Since the hardware aspect is not explicitly considered in design, the system must be able to scale according to hardware requirements and performance, so that a multitude of processors and hardware topologies can benefit from the design. The system is targeted for ARM-based MCUs, especially for STM Cortex M4 and M7, which both have FPUs available. However, the system should also efficiently function on other MCUs with included FPUs. Care must be taken when writing mathematically-intensive code, to ensure optimal use of the hardware, such as leveraging FPU instructions, avoiding data duplication, utilising cache memory etc.

The technical requirements for the system are as follows:

1. An audio system that can generate buffers of a specified size of stereo audio data efficiently.
2. An arbitrary sampling rate may be specified. Common audio sampling rates [6] such as 44.1, 48 and 88.2 kHz can be used.
3. Care must be taken to avoid or minimise aliasing.
4. An arbitrary polyphony number (maximum number of notes) may be specified. A ”generator” refers to the object that is generating the audio for a single note.
5. When the number of active notes exceeds the polyphony, the generator containing the oldest playing note must be re-triggered.
6. The system managing the generators must do so efficiently in O(N) time, where N refers to the polyphony number.
7. FPU/CPU intensive operations such as divide or modulus operation must be avoided at all costs, if possible.
8. The DSP chain must consider efficiency, and simplify processing as far as reasonably practical.
9. The core software must be in written in C, to allow compilation for any MCU, and to have full control over memory usage and predictability in the assembly code. Therefore, no object-orientation will used, and functional data manipulation will be the prime consideration.
10. Processing speed takes priority over memory consumption, as far as reasonably practical.
11. An arbitrary wavetable buffer size can be specified, to provide control over the memory consumption and maximum harmonic content of a waveform.
12. All DSP must be performed with single-precision floating-point operations, to allow for full 24-bit audio resolution for DAC conversion, which many SAI codec ICs support [7].
13. Filtering must be computationally efficient. This implies that IIR filtering must be used, since human hearing is insensitive to phase shifts of higher harmonics within the filtered signal.

1.4. Summary of work

The following items summarise the work that has been performed in this thesis:

1. A computationally efficient method of doing table-lookups with linear interpolation for periodic signals is created.
2. The effect of frequency scaling and linear interpolation is modelled in the frequency domain, forming the bases of predicting the effect of higher-order interpolation filters for wavetable frequency scaling.
3. A wavetable schema is designed to avoid aliasing for higher frequency notes.
4. FM and inter-wavetable interpolation was explored.
5. The design and efficient coefficient computation for 5 IIR filters is done.
6. LUTs for waveshaping and a technique for anti-aliasing is done through numerical analysis.
7. An ADSR state-machine that outputs a piecewise-exponential is created.
8. A way to create a note with stereo width is detailed.
9. An efficient way for managing generator objects is devised for managing note on/off triggers.

1.5. Report overview

Literature overview

The history and operation of synthesis is explored in this chapter. A detailed explanation of synthesis concepts and components are detailed by using existing synthesisers and/or eurorack modules as examples. The required musical, audio and mathematical knowledge for this thesis is detailed here. This chapter only covers existing techniques and knowledge, with some elaboration when necessary.

Design

This chapter details the design of the audio system. A top-down system specification with bottom-up component synthesis methodology is applied here. All the mathematics and self-developed techniques, and the application of existing techniques are documented in this chapter.

System testing

Each component is quantitatively tested in this chapter, along with a full system test that demonstrates most of its ability. Due to the number of user parameters and the complexity of the resulting audio, a full system test that demonstrates all the functionality is not possible. The nature of this system ensures correct operation if all the components are properly tested, since chaining the components together correctly will not cause system failure. It is up to the user to set parameters. This is similar to how a modular synthesiser system operates: correct system operation depends on the correct operation of the modules. Instead, a musical test is done in appendix B.6, with some qualitative comments on the audio.

Conclusion

The design and its achievements are discussed, with reference to the system specification. Possible improvements and further applications is detailed.

Chapter 2
Literature overview

2.1. A brief history of musical synthesisers

An instrument usually described as a “synthesiser” or “synth” is any electronic device or software that can generate audio signals. This can be done through a variety of techniques that have been developed and have evolved during the 20th and 21st centuries.

A non-keyboard style of synthesis - modular synthesis - was popularised by the companies Moog and Buchla in the 1960s [8]. Analogue electronics were used for synthesis where multiple modules generate control voltages in tandem to modulate parameters. Common building blocks are VCOs, envelope generators (ADSR), VCFs, VCAs, sequencers, wave-shapers, and noise generators. These are still the fundamental aspects of most synthesisers to date. Emulation or cloning of original Moog or Buchla hardware such as the 4-pole ladder VCF is still sought after in the current commercial market [9].

In contrast, modern Eurorack modules (a standardised modular hardware and electronics specification) offer a wide variety of complex options, which are often analogue or digital-analogue hybrid . An example is the Make Noise MATHS [10] module which is very popular in the modular synth community [11]. It provides features such as amplification, integration, summation, function generation, and is considered an essential module by many influencers.

Due to the complexity of analogue electronics, most synths were still monophonic in the 1960s and 1970s, or had very limited polyphony. Each extra note required duplication of electronics, which, in turn, requires more fine-tuning. The introduction of digital technology in the 1980s allowed for more flexible polyphony at affordable prices. The Yamaha DX7 is an incredibly well-known early digital synthesiser released in 1983 [12], which used FM-synthesis (see section 2.2).

The introduction of more powerful computation led to the development of software synths and VSTs for DAWs which use a variety of synthesis techniques such as FM, additive synthesis, subtractive synthesis, physical modelling, and wavetable synthesis. Wavetable synthesis is very popular in all music genres and sound design for film. Some of the most popular instrument VSTs used are Serum by Xfer Records [13], Massive by Native Instruments [14], and PIGMENTS by Arturia [15]. The aforementioned VSTs focus on wavetable synthesis with sampling, filtering, parameter modulation and FM capabilities.

2.2. An overview of synthesis techniques

Most techniques [16] are often combined in commercial products and operate in a similar way. The similarities, differences and operation principles will be explored in this section . This is will also explore why wavetable synthesis can be considered the most flexible and computationally robust technique.

2.2.1. Additive synthesis (Fourier synthesis)

The principle of operation is based on the harmonic series of a time signal:

$$y(t) = \sum_{n=0}^{\infty} a_n sin(\omega nt + \phi_n)$$

Various sinusoids are added together with different amplitudes and phases to produce a signal. The amplitudes and phases and frequencies may be time varying as well, which ties into physical modelling techniques that accounts for the time dependent timbre of most instruments.

Computing and adding many sinusoids in real-time can be computationally expensive – which can be reduced with a sinusoid LUT, as is directly done in wavetable synthesis. If time-varying amplitudes and phases are not present, additive synthesis can be completely replaced by wavetables (LUTs).

2.2.2. Subtractive synthesis

This technique is very simple and is possible in most synthesisers. It requires a harmonically rich source signal (generated by any means, such as direct computation, LUTs or analogue electronics) like a square wave:

$$y(t) = A \cdot sgn(sin(\omega nt + \phi_n))$$

It consists of odd harmonics with amplitudes $a_n \propto \frac{1}{n}$. The source signal is then passed through a filter to further shape the harmonics. Any filter can be used. Time varying filters with modulated parameters are usually prevalent. This option is almost always present and/or possible to achieve in most synthesisers that offer filtering capabilities. Many products [17] usually offer a selection of base waveforms which often includes most of or all the basic waveforms (sinusoid, triangle, sawtooth and square).

2.2.3. FM synthesis

This technique uses the same principles as FM for data communication, except in the audible frequency range. The resulting waveform is of the form:

$$y(t) = g(\omega(t))$$

where $g(\omega(t))$ is a periodic function with time-dependent frequency ω. This technique can produce unique and interesting results depending on the functions chosen for g and ω.

The choices offered for chosen for g and ω are product dependent but can often include the basic waveforms for g and ADSR envelopes and LFOs for ω. Multiple oscillators modulating each other's frequency, often in a coupled or recursive manner, is common, as in the stock Ableton VST plugin Operator [18], which is a FM-centric VST. Many non-FM-centric synths also offer a vibrato feature, which requires the use of dedicated vibrato LFO that slightly modulates the source signal's frequency. This is present in VSTs such as Omnisphere 2 by Spectrasonics [19], which is wavetable and sample-based.

2.2.4. Physical modelling

This method involves simulating the sound source of interest. It is usually separated in continuous models for bowed or blown instrument or impulsive models such a struck or picked instruments. A variety of methods can be used, such as IR modelling, analytical simulation (differential equations), frequency domain modelling as mentioned under additive synthesis, and waveguide synthesis such as the Karplus-Strong plucked string algorithm [20] [21]. This type of synthesis is not relevant to the topic of this thesis.

2.2.5. Sampling

Sampling synthesis is the technique of using pre-recorded audio samples to reproduce sounds. An example would be to record every key of a piano at different volumes and then assigning a sample to trigger when conditions are met [21]. The Kontakt player by Native Instruments [22] is a popular sample player plugin into which third-party sample libraries can be loaded into to reproduce high-quality and realistic audio. High quality samples often take enormous amounts of effort to make, which results in a high commercial price point as can be seen in the Omnisphere 2 VST [19] and the Spitfire Audio Kontakt libraries [23].

It is very similar to wavetable synthesis, where a predefined buffer (LUT) is used to generate sound. However, sampling often uses large buffers that are not necessarily intended to reproduce a periodic waveform (but sometimes do for continuous sound produced by instruments such as flutes). Samples and wavetables can be manipulated and modulated in the same way. This technique is computationally efficient but may require a large amount of memory to store the samples – often in the order of gigabytes, as for Kontakt libraries.

2.2.6. Wavetable synthesis

Wavetable synthesis [16] is a very powerful, efficient and popular technique used in many modern synthesisers, which includes VST instruments, keyboards and eurorack modules.

A periodic waveform is stored in a table , which is sampled at a specific rate (see sections 3.4.1 and 3.4.2). A variety of these tables can be stored (even created by the user as in Serum [13]) and manipulated by interpolating between wavetables or manipulating the wavetables themselves, such as with folding or adjustable duty cycle. This technique allows for complete freedom in parameter modulation. This includes techniques such FM. Figure A.2 in the appendix shows an example of a wavetable VST synthesiser [18].

2.3. Basic modular synthesiser building blocks

This overview focuses on modular synthesiser building blocks directly but is relevant to most forms of synthesis (especially in this thesis) since most standard modern synthesis products is based on the building blocks popularised by Moog and Buchla [24]. Example modules will be shown, discussed, and compared to features present in commercial wavetable synthesisers and features to be considered for design in this thesis. Refer to appendix B.2 for an explanation of a basic modular synthesiser setup. Figure A.1 in the appendix shows the modules that is used as examples [17] in this section.

2.3.1. The VCO

VCO modules commonly include 1V/octave CV inputs for frequency and provide the basic waveforms as outputs either separately via a switch or simultaneously. They can be analogue or digital in nature and can use a variety of synthesis techniques to generate their waveforms. They often come with the ability set the offset tuning voltage and can be used to create FM signals through control voltages. Extra features such as wave folding are sometimes also present.

The Doepfer A-111-3 Micro Precision VCO/LFO [25] is an analogue VCO that can also operate in LFO configuration, either with a linear or exponential voltage control. Sync (for phase/frequency syncing) and PWM CV inputs are also available. All the basic waveforms are present, except for the sinusoid which is notoriously difficult to generate with analogue electronics, which is commonly implemented by a high-Q unstable filter.

2.3.2. The VCF

The VCF is an incredibly important module that forms the basis of subtractive synthesis techniques. Most synths also offer filtering capabilities, such as the widely used Nord Stage 3 [26].

Filters can come in many types, often designed with unique characteristics. This can include special control voltage behaviour, feedback path saturation to limit resonance while adding additional harmonics, or the ability to achieve exceptionally high Q values that cause purposeful instability that allow filters to also function as a sinusoidal oscillator (which many VCOs do not generate).

Thus, filters for musical applications are usually not designed to be as “clean” and stable as possible. Instead, they focus on usability and uniqueness. Filter types can include a switchable LPF, BPF or HPF mode, a ladder filter, 12dB/octave or 24dB/octave varieties and a state variable filter configuration.

The IntelliJel UVCF [27] is popular state variable filter that simultaneously outputs a 2-pole low-passed, 2-pole high-passed and 1-pole band-passed signal which has a cut-off that can be modulated by 2 separate 1V/octave control voltages. It can also be set to have a high Q-value so that it can act as a sinusoidal VCO due to filter instability.

2.3.3. The VCA

The VCA has the primary purpose of performing the multiplication of signals for uses in AM and otherwise. It acts as an amplifier with a voltage controllable gain. It is often used in conjunction with an LFO to create a tremolo effect or with an ADSR envelope to shape the transient of signal to emulate bowing or plucking and removing clicks and pops that can occur with the immediate triggering of signal. Many VCOs only output a continuous signal. Hence, a VCA is required to mute any oscillators that are not triggered. The ring modulation effect can also be achieved by multiplying 2 signals in the audible frequency range together.

The MFB VCA [28] is module that has 3 different inputs and 2 CV inputs that modulate the gain. The operation of the various inputs is specific to this module and out of the scope of this thesis.

2.3.4. The ADSR envelope

The ADSR envelope is a critical component in synthesis used to achieve realistic sounds. See figure It is often used to modulate filter cut-off to allow for dynamic subtractive synthesis. It is also used for AM to emulate the natural attack and decay characteristics of real instruments. It can emulate plucking, strumming, bowing, and blowing techniques found in real instruments. It can also be used in FM to recreate the typical pitch modulation found when striking percussive instruments. Due to the logarithmic nature of human hearing [29], an exponential shape is required of the envelope so that a linear change in volume is perceived.

ADSR envelopes are available in most wavetable synthesisers for parameter modulation, such as Serum, Massive and Ableton’s stock Wavetable VST instruments. Many keyboards also include this feature, such as the Nord Stage 3 [26]. The envelope consists of 4 phases. The curve is initiated with a gate “on” trigger after which a rising function is started. Once a threshold is reached, determined by the attack time, the decay state is activated. The decay is specified by a decay time parameter. The function decreases until a sustain level is reached, which is a parameter set by the performer. The sustain level is usually less than the peak achieved by the attack phase. The sustain phase remains constant until the gate signal changes state to indicate an “off” trigger, initiating the release phase. The release phase is a decreasing function that decays until zero is reached (or close to zero in the case of an RC circuit), determined by a release time parameter. The A, D and R phases are usually exponential functions implemented by an RC circuit. This is well suited for AM and FM, since octaves are exponential in nature (doubling in frequency) and human hearing is logarithmic in nature [29] – an exponential volume change is perceived as linear.

The Doepfer A-140 ADSR Envelope Generator [1] is a classic envelope generator with a gate CV input, an envelope output, and a negated envelope output. It also has a retrigger input that allows the “on” trigger to occur again, reinitiating the attack phase, independent of the current phase of the envelope. A typical ADSR envelope can be seen in figure A.3 in the appendix.

2.4. Prerequisite knowledge

2.4.1. Equal temperament tuning

Various tuning systems have existed since the inception of standardised instruments. A common problem with dividing the octave into 12 notes (used in most Western music) is the inconsistent ratios between notes when a different root key is chosen. This problem is the result of using the natural harmonic series to define the ratios between pitches. I.e, a perfect fifth is the ratio 3:2, which was used to define the Pythagorean tuning.

There is no way to tune 12 notes in a scale that will result in equal integer ratios for all intervals across all notes and octaves. Thus, equal temperament tuning was introduced to solve this problem [30].

This tuning system uses $\sqrt[12]{2}$ as the relationship between semitones, resulting in equal ratios for all interval across all octaves. The standard for tuning is defined by the frequency of concert A (A4) to be 440 Hz. Consequently, the n'th semitone after A4 is $440(\sqrt[12]{2})^n$, and the k'th semitone before A4 is $440(\sqrt[12]{2})^{-k}$. Furthermore, each semitone is divided into 100 equal steps, which is known as cents, with each cent differing from the next by a ratio of $\sqrt[1200]{2}$. This is a measure of the intonation of note.

A frequency ratio of $c \in \mathbb{R}$ cents, can be split into a combination of semitones ($x \in \mathbb{Z}$) and cents ($y \in [0, 100)$) as shown in equation 2.1.

$$2^{\frac{c}{1200}} = 2^{\frac{x}{12}+\frac{y}{1200}} = 2^{\frac{x}{12}} \cdot 2^{\frac{y}{1200}} \tag{2.1}$$

Where $x = \lfloor \frac{c}{12} \rfloor$ and $y = c - 12x$. Using this type of decomposition for a frequency ratio, allows for easy table lookups into semitone and cent LUTs to allow accuracy in frequency scaling on the cent level, i.e. when $y \in \{0, 1, ..., 99\}$. Linear interpolation can then approximate these ratios between integer cent values. See code listing D.1 in the appendix for the C implementation.

2.4.2. Audio

Stereo

Most consumer audio is in a stereo format, i.e. having a left and right audio channel. It can then further be converted to mid (M) and side (S) channels using equations 2.2 and 2.3 [31]. The mid channel can be considered as the "mono'd" version of stereo audio, which is used by devices such as phone speakers, which cannot play stereo audio. The side channel can be considered as the signal containing all the stereo information, i.e. a measure of audio width. Audio processing is sometimes done on the mid and side channels instead of the left and channels. After processing, the left and right channels are reconstructed.

$$M = \frac{L+R}{2} \tag{2.2}$$

$$S = \frac{L-R}{2} \tag{2.3}$$

Note that if $L = R = y[n]$ (a mono signal played through both stereo channels), then $S = 0$, $M = y[n]$, implying that no stereo information is present. Using equations 2.2 and 2.3, we can then reconstruct the L and R signals using:

$$L = M + S \tag{2.4}$$

$$R = M - S \tag{2.5}$$

Quality

A variety of digital properties can determine the quality of the audio that is streamed. Prime considerations are sampling rate and bit depth. There are other considerations, such as dithering and encoding (PCM, DPCM, etc.) [32].

The sampling rate determines the bandwidth of the audio, as per the Nyquist criterion. Since human hearing is restricted to 20 Hz to 20 kHz [29], sampling rates for high-fidelity audio often exceed 40 kHz. Common rates are 44.1, 48 and 88.2 kHz [6]. The sampling rate is often higher than required, to avoid aliasing when processing the signals. The signal is sometimes up-sampled (often through linear interpolation) to double or quadruple the sampling rate before processing to combat aliasing.

The bit depth refers to the amount of bits used to store the audio data. The way it is stored depends on the encoding, which will yield different quantisation error probability distributions. The bit depth determines the noise-floor of the signal, expressed via SQNR. Typical bit-depths are 16, 24 and 32 bits, with SQNRs of 96.33, 144.49 and 192.66 dB respectively [33].

2.4.3. Prototype IIR filters

A popular technique for designing an IIR filter is by using a continuous prototype filter $H(s)$ with critical frequencies $\omega_c = 1$ rad/sec, by utilising the bilinear transform and fequency scaling [2].

A prototype filters are converted to the Z-domain to be in the form

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}} = \frac{N_0 + N_1 z^{-1} + N_2 z^{-2}}{1 - D_1 z^{-1} - D_2 z^{-2}} \tag{2.6}$$

where N_0,N_1,N_2,D_1,D_2 are the coefficients normalised by a_0. D_1 and D_2 are negated.

Converting back to the discreet time domain, the output of the filter ($y[n]$) can be calculated as

$$y[n] = N_0 x[n] + N_1 x[n-1] + N_2 x[n-2] + D_1 y[n-1] + D_2 y[n-2] \tag{2.7}$$

The negation of D_1 and D_2 proves useful in this circumstance, since now we can take advantage of the typical multiply-and-accumulate FPU instruction that many processors offer.

A list of substitutions to convert from the S to Z domain using the bilinear transform is shown in table C.1 in the appendix. The table also takes frequency mapping into account. The cut-off frequencies are mapped from 1 rad/s in the prototype filter to $\omega_0 = 2\pi\frac{f_0}{f_s}$ rad/sample (normalised digital frequency).

Chapter 3

Design

In this chapter, we apply bottom-up synthesis for designing our system, by designing, modeling and deriving behaviour of the lowest-level components first. When implementing code for this section, special consideration is made for compilation and FPU instructions. This will not always be discussed. See table C.3 in the appendix for a list of ARM FPU assembly instructions.

Only the most important code implementations are shown in this section. If implementation is trivial or large, the C code is listed in appendix D. Enough information will be by provided by the system block diagrams and accompanying equations to make other implementations possible.

This system was first partially implemented for integer arithmetic using Q-numbers, but was rewritten for floating-point operations after encountered problems, and extra design and implementation considerations. See appendix B.3 for details.

All code is publicly available at `https://github.com/marcorad/skripsie`.

3.1. DSP pipeline

The system consists of a number of functional components which have been detailed in section 1.3.2. Figure 3.1 shows a high-level description of all of the components and the processes that they must perform. The components are numbered according to a top-down design approach, where number 1 refers to the highest level of operation between all components. Components are only dependent on other lower-level components.

3.1.1. Component definition

Each component can be defined in terms of its lower-level component dependencies, the required functionality and its inputs/outputs.

1. **Generator manager**: this component manages the generators, by efficiently choosing an available generator, or the oldest playing generator if none are available. It receives note on/off requests, associates it with a generator, and does all triggering and configuration (such as frequency) that is required. The active generators can also be sampled, upon a audio buffer request. In here, generator samples are summed, scaled and clamped to a range [-1, 1]. This component uses a MIDI-note hashtable, and a generator queue and stack, to manage note associations and active/inactive generators.

2. **Generator**: this component represents the collection of data that is required to produce stereo audio samples for a single triggered note. This component is responsible for retrieving and blending wavetable samples, applying waveshaping (and the anti-aliasing required), sampling the ADSR envelopes for volume and filter modulation, triggering the recalculation of filter coefficients and applying the filtering.

3. **Wavetable**: this component is responsible for all LUT lookups and inter-LUT interpolation, while also managing the periodicity (as per subsection 3.3), frequency and harmonic content of a periodic waveform. It can receive FM input for vibrato. It produces mono samples that is to be used in conjunction with other wavetables in stereo blending.

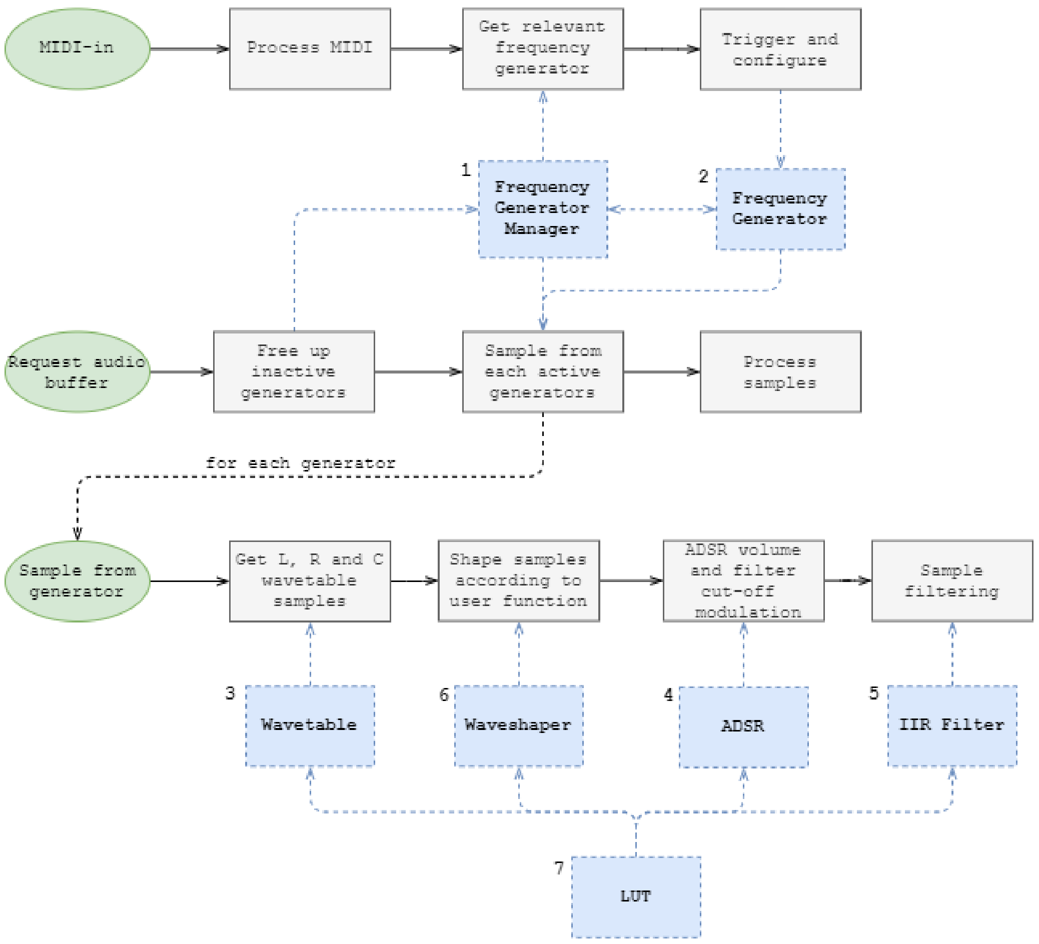

MIDI-in
Process MIDI
Get relevant frequency generator
Trigger and configure
1
Frequency Generator Manager
2
Frequency Generator
Request audio buffer
Free up inactive generators
Sample from each active generators
Process samples
for each generator
Sample from generator
Get L, R and C wavetable samples
Shape samples according to user function
ADSR volume and filter cut-off modulation
Sample filtering
3
Wavetable
6
Waveshaper
4
ADSR
5
IIR Filter
7
LUT

Figure 3.1: High-level system description.

4. **ADSR**: this component produces mono samples from an ADSR curve, using table lookups into an exponential function LUT. It is a state-machine that manages the piece-wise exponential ADSR curve, given ADSR parameters. It can be queried to determine whether the release phase has finished. It can receive on and off triggers, and can be re-triggered to the attack phase, at any moment, if necessary.

5. **IIR filter**: this component contains all the functions from different filter types that can calculate the IIR coefficients. It stores all necessary sample delays, for each filter instance, that are required to perform filtering. It can apply filtering to an input signal, and has the ability to be reset (sample delays set to 0).

6. **Waveshaper**: this component can apply a waveshaping function (either sin, tanh, or none) to an input. It does not handle anti-aliasing (the generator does). It is responsible for doing lookups into the function LUTs and managing periodicity with the sin lookup, and out-of-bounds tanh lookups.

7. **LUT**: this component does a singular table lookup, with linear interpolation, into any array, without checking for an out-of-bounds index. Ensuring a within-bounds lookup index the responsibility of the parent components. This component is the core of the audio engine, and must be fast and efficient.

3.1.2. Global user parameters

From the functional specification in section 1.3.2, we can determine all the global parameters that is applicable to every playing note. These parameters must be separated from the data contained within the components, so that it can easily be changed during an audio stream, to provide the performer with continuous feedback.

Table 3.1 summarises all the global parameters, indicating to which category of processing they are relevant, and the units they must be stored in. Note that the attack, decay and release parameters are stored in terms of digital frequency, and not seconds. This is so that LUT lookup pointers can easily be updated. All frequencies in this system are stored in digital frequency, as to remain sampling-frequency-agnostic.

Table 3.1: All global parameters

Category	Parameter	Unit(s)
Volume	Envelope: attack, decay, release	cycles/sample
	Envelope: sustain	-
Filtering	Envelope: attack, decay, release	cycles/sample
	Envelope: sustain	-
	Relative frequency start	-
	Relative frequency end	-
	Q	-
	Type	-
Detune	Amount	cents
	Stereo width	-
	Volume	-
Vibrato	Frequency	cycles/sample
	Intensity	cents
Waveshaping	Input gain	-
	Type	-

3.2. Top-level system in MCU implementation

The top-level system is the entry-point of MIDI messages and is illustrated in figure 3.3. In this part of the system, MIDI commands are parsed and added to a queue. An audio buffer request is made, where a fixed amount of stereo samples are retrieved from the generators and stored in a buffer. This is the system that is to be implemented on a MCU, and is thus not within the scope. Instead, a SMF will be used to provide input (see appendix B.6) to demonstrate functionality using musical test, which will output sound to a wav file.

Figure 3.2 shows the legend that should be used to interpret the block diagrams shown in this chapter.

The note queue is only processed at the beginning of a buffer request, thus adding latency into the system, that the performer will experience. The latency (l) that is added to the system is is a function of the size of the audio buffer (N), specified in number of stereo samples and the sampling rate (f_s).

$$l = \frac{N}{f_s} \tag{3.1}$$

The added latency must minimal to ensure no effect on live performance. Typically, latencies of 10 ms to 15 ms are acceptable and unnoticeable to performers. Thus, at a sampling rate of 44.1 kHz, a maximum size of 661 stereo samples are acceptable for the buffer.

Also note that a note on/off trigger is restricted to a minimum time equal to the latency, otherwise a note on/off trigger pair will cancel out, since they are both processed at the same time. Furthermore, extra equipment in the signal processing chain, such as external effect and amplifiers, may add more latency. Thus, it is better choose an audio buffer size of around 2 ms, so that the instrument can still be real-time-usable in conjunction with an external signal chain.

The choice of buffer size is not relevant in this thesis, except for note triggering time, since system testing is not done for a real-time application. The system is designed to be compatible for any choice of buffer size and

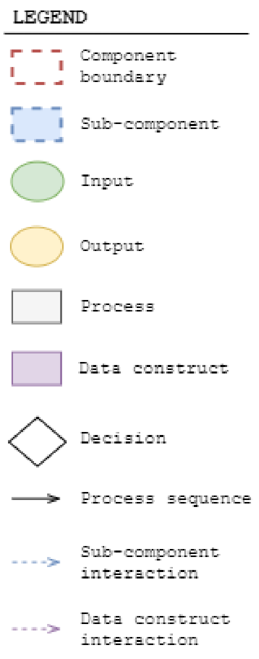


Figure 3.2: System block diagram legend.

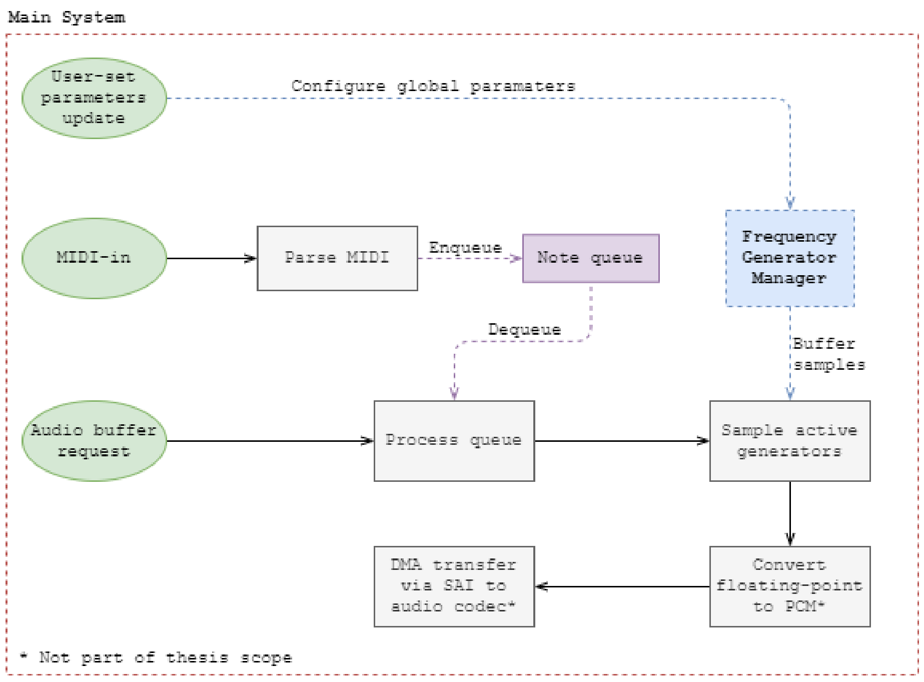

Main System
User-set parameters update
Configure global paramaters
MIDI-in
Parse MIDI
Enqueue
Note queue
Frequency Generator Manager
Dequeue
Buffer samples
Audio buffer request
Process queue
Sample active generators
DMA transfer via SAI to audio codec*
Convert floating-point to PCM*
* Not part of thesis scope

Figure 3.3: The top-level system block diagram.

sampling rate. Choosing these must be done with the hardware platform in mind. Generally, a higher sampling rate will reduce aliasing, which must be leveraged if the hardware has enough processing power.

3.3. LUT

3.3.1. Linear interpolation

Linear interpolation is process of defining a function in terms of points, and then connecting the points in a piecewise-linear fashion. This is especially relevant for LUT lookups, when a value between points is required for better accuracy. This is the core of the system. Figure 3.4 shows the block diagram for this system component.

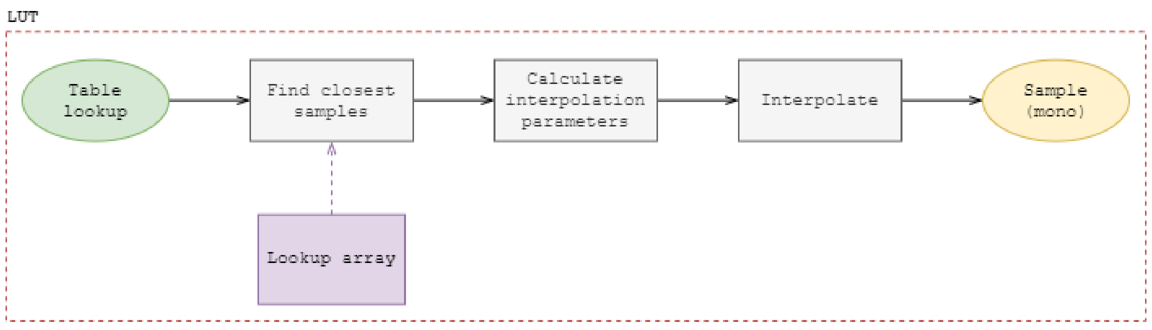


Figure 3.4: LUT system block diagram.

The linear interpolation (lerp) function is defined as follows:

$$\operatorname{lerp}(x_1, x_2, \delta) = x_1 + \delta(x_2 - x_1) \tag{3.2}$$

Where x_1 is the starting point, x_2 is the endpoint, and $\delta \in [0, 1]$ is the interpolation distance.

Assuming that we have a LUT ($L[n]$) storing N samples indexed with $n < N$, $n \in \mathbb{N}_0$, we can find a linearly interpolated point $p(i)$ at position $0 \leq i < N-1$, $i \in \mathbb{R}$ using equation 3.2.

$$p(i) = \mathrm{lerp}(L[\lfloor i \rfloor], L[\lfloor i \rfloor + 1], \{i\}) \tag{3.3}$$

If periodic behaviour is required from the LUT, we can wrap i around when it exceeds the limits, either through using the fractional function or the modulus function for real numbers.

The modulus operation can be defined through floored division as shown in equation 3.4, which can also provide an extension of the function into the real numbers for $a, r, n \in \mathbb{R}$.

$$a \equiv r \pmod n \Leftrightarrow r = a - n\lfloor \frac{a}{n} \rfloor \tag{3.4}$$

Equation 3.4 can be implemented efficiently in code if n is a power of 2, and $a \in \mathbb{N}_0$. Listing 3.1 shows the C implementation for this function. Note that the use of the “inline” keyword is a C++ compiler directive, but is only used to increase the speed of the program by avoiding branch penalties when calling the function. We thus restrict all periodic LUTs to have size that is a power of 2.

```c
inline uint16_t fast_mod(uint16_t x, uint16_t mod) {
    return x & (mod - 1);
}
```

Listing 3.1: Fast modulus for a power of 2

Listing 3.2 shows the implementation of equation 3.3 and figure 3.4 in C, where special care has been taken to ensure that if $i \in [N-1, N)$, the interpolation will be performed between the samples $L[N-1]$ and $L[0]$. This is required for interpolating periodic LUTs.

```c
inline float lut_lookup(float lut[], uint32_t lut_size, float i) {
    uint32_t floor_i = (uint32_t) i;
    float delta = i - (float)floor_i;
    float x1 = lut[floor_i];
    float x2 = lut[fast_mod(floor_i + 1, lut_size)]; //wraps i to 0 if i = lut_size-1
    return lerp(x1, x2, delta);
}
```

Listing 3.2: LUT lookup with linear interpolation

The fractional function can also be used to ensure periodicity. Equation 3.5 shows the definition of the fractional function and its range.

$$\{x\} = x - \lfloor x \rfloor \ , \ \{x\} \in [0,1) \ \forall \ x \in \mathbb{R} \tag{3.5}$$

The fractional function is periodic with a frequency of 1 Hz, and contains a straight line starting at (0,0) and approaching (1,1) over $x \in [0,1)$. Thus, we can create a function ($W(t)$) that wraps the index so it repeats with a period of p, with a shifting factor (k), and an amplitude (A), shown in equation 3.6. This can be used to easily wrap the index into a LUT of size A, with p defining the input range of t, with k setting the starting lookup index ($W(t=0)$).

$$W(t) = A\{\frac{1}{p}(t-k)\} \tag{3.6}$$

Note that when $k = \frac{p}{2}$, then a period of $W(t)$ is contained in $t \in (\frac{-p}{2}, \frac{p}{2})$, having a range $W(t) \in [0, A)$, and a starting lookup index $W(t=0) = \frac{A}{2}$.

3.3.2. Constructing the basic waveform LUTs

In this section we will investigate how to generate a Fourier series of the basic waveforms (square, triangle and sawtooth) for LUT look-ups and explore memory optimisation techniques. The waveform of interest will be represented as the following series:

$$y[n] = \sum_{k=1}^{K} a_k \sin(2\pi k \frac{n}{N}) \tag{3.7}$$

Where k is the wavenumber, a_k is the k'th amplitude coefficient, K is the number of harmonics, and $N = 2^b$ is the size of the LUT that will store a complete period of the waveform, i.e. $0 \leq n < N$, $n \in \mathbb{N}_0$.

With reference to equation 3.7, the coefficients a_k can be derived as follows:

1. Square:

$$a_k = \begin{cases} \frac{1}{k}, & k \equiv 1 \pmod 2 \\ 0, & k \equiv 0 \pmod 2 \end{cases} \tag{3.8}$$

2. Triangle:

$$a_k = \begin{cases} \frac{(-1)^{\frac{k-1}{2}}}{k^2}, & k \equiv 1 \pmod 2 \\ 0, & k \equiv 0 \pmod 2 \end{cases} \tag{3.9}$$

3. Sawtooth:

$$a_k = \frac{(-1)^{k+1}}{k} \tag{3.10}$$

Equations 3.8 and 3.9 imply only odd harmonics, hence for harmonic index (see equation 3.14) $i = 0$, it will represent a simple sinusoid.

Furthermore, since equation 3.7 is a linear combination of harmonic sinusoids with no phase-shift, and $a_k > 0$ for all odd k, $a_k \leq 0$ for all even k in equations 3.8 to 3.10, we can conclude that all basic waveforms and their harmonics are in phase. This makes it possible to interpolate between wavetables (as seen in Ableton's Wavetable VST in figure A.2) without phase cancellation. Furthermore, to allow for range predictability, the samples are normalised so that the range falls within $[-1, 1]$.

The sinusoid is trivial to construct, since it does not contain any other harmonics. Thus, we can store the 4 basic waveforms in a $4 \times K \times N$ array. "4" corresponds to the basic waveforms, ordered by increasing harmonic content (sine, triangle, sawtooth, square); K refers to the amount of copies with different harmonics we want to store of each waveform; $N = 2^b$, $b \in \mathbb{N}$ is the amount of samples per waveform period. The 3D array allows for code simplicity when each note requires an harmonic index and inter-wavetable interpolation (see section 3.4).

In this case, we are duplicating the sinusoid LUT K times. However, we prioritise speed and code simplicity over memory consumption . Adding code to deal with the sinusoid specifically could incur branch penalties, and will require a special array for the sinusoid, which will increase code complexity.

3.4. Wavetable

This component is responsible for generating a specific frequency of an interpolated waveshape using the basic waveform LUTs. Figure 3.5 shows the block diagram.

Each wavetable requires its own data structure to store relevant values, which includes stride and base stride (stride $\propto \eta$, see equation 3.13), phase (ϕ), and harmonic index (i). Base stride refers to the center frequency of the wavetable, whereas stride includes FM.

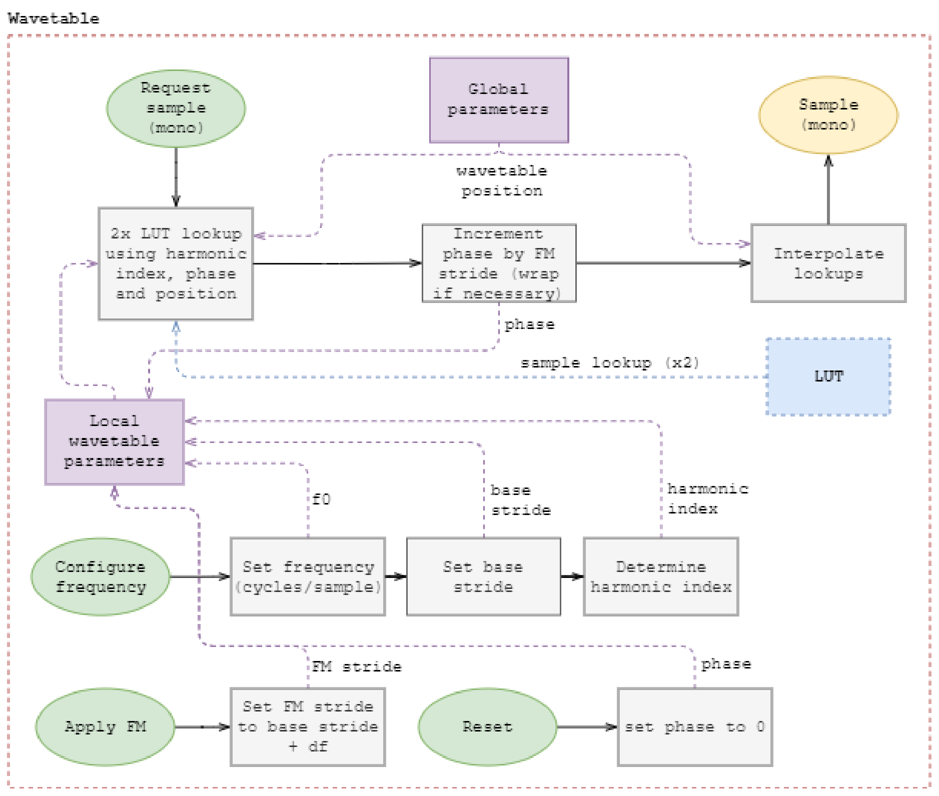


Figure 3.5: Wavetable system block diagram.

Each sample request results in the appropriate wavetable lookup and inter-wavetable interpolation. The phase is incremented after each request. The frequency must also be configured before use, required to calculate the harmonic index. A reset trigger also resets the phase. The rest of this section provides further explanation, derivation and insight into these parameters.

3.4.1. Modelling wavetable frequency conversion

This section focuses on modelling wavetable sampling and determining the effects of linear interpolation and frequency scaling in the frequency domain.

Suppose we want to store a periodic signal $y[n]$ in a buffer consisting of 2^b samples, where $b \in \mathbb{N}$. The fundamental frequency of the buffer as normalised digital frequency (cycles per sample) is thus 2^{-b}.

If we require the buffer to store H harmonics including the fundamental, we use the Nyquist frequency to determine the maximum number of harmonics the buffer can store. This can be specified by the user of the code, according to the memory restrictions of the hardware platform.

$$H_{max} 2^{-b} = 0.5 \Rightarrow H_{max} = 2^{b-1} \tag{3.11}$$

Therefore, a bigger buffer size results in the ability to store more harmonics.

Now we consider the follow process shown in figure 3.6 to scale a sampled signal's frequency ($y[n]$) by a factor of $\eta = \frac{M}{L}$. The signals in the frequency conversion process is modelled as $\hat{y}[n] = (y[n]_{\uparrow L} * h[n])_{\downarrow M}$.

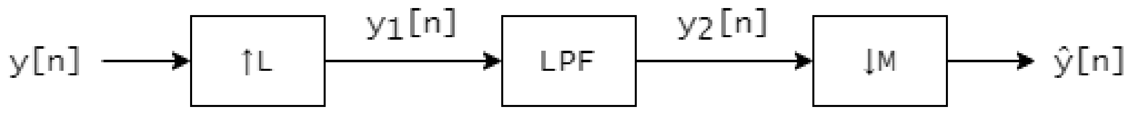


Figure 3.6: Frequency scaling using upsampling and downsampling

As per the scaling theorem [34], the frequency axis of $DTFT\{y_1[n]\}$ contracts by a factor L. A LPF is used to remove unwanted copies of the spectrum, which would result in aliasing if the frequency axis is expanded by a factor M through downsampling.

There are many choices for the LPF, but for arbitrary frequency scaling, M and L will become large to achieve close approximation for any real number. The simplest filter choice would be the linear interpolator, which is a FIR filter characterised by its impulse response [34]:

$$h[n] = \text{tri}[\frac{n}{L}] = \frac{1}{L}\text{rect}[\frac{n}{L}] * \text{rect}[\frac{n}{L}]$$

Taking the DTFT,

$$H(f) = \frac{1}{L}\text{DTFT}\{\text{rect}[\frac{n}{L}]\}^2 = \frac{\sin^2(L\pi f)}{L\sin^2(\pi f)}$$

Note that $H(f) = 1$ for $L = 1$. For $L \neq 1$, it has zeroes at $f = \frac{p}{L}, p \in \mathbb{N} \setminus \{0\}$. Figure 3.7 shows the effect of the linear interpolation process in the frequency domain. The bandwidth of $y[n]$ is represented as a triangular pulse.

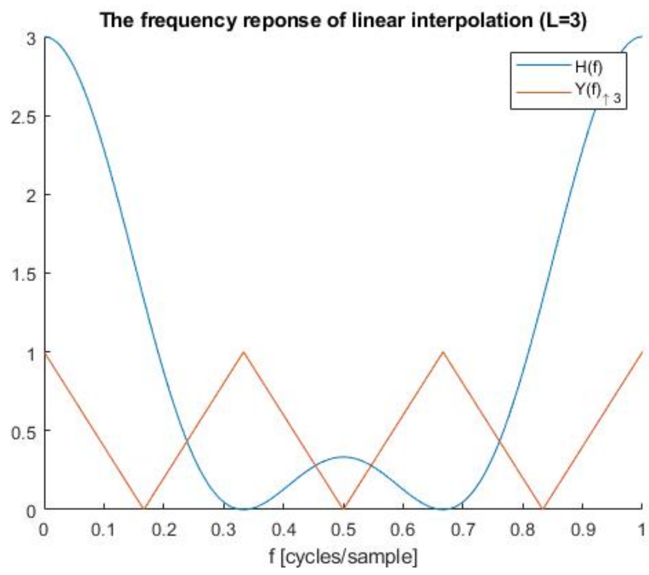


Figure 3.7: The effects of linear interpolation in the frequency domain.

With reference to figure 3.7, notice that $H(f)$ has nulls at the DC component of the upsampled signal spectrum. All the frequencies that are close to DC, i.e. lower frequencies, have high attenuation. Thus, when the sampling rate is increased, the spectral distortion is not affected, since the wavetable fundamental is fixed at 2^{-b} cycles/sample. Increasing b, however, will lower the fundamental frequency and result in less harmonic distortion when interpolating lower frequencies.

The linear interpolation filter has a fixed cutoff which is a function of L. When the frequency is scaled up, i.e. $M > L$, the original spectrum with a full bandwidth will alias since the linear interpolation filter does not account for this. To combat this issue, we can use a variety of buffers for a single waveform, each containing a different number of harmonics, thus band-limiting the signal to combat aliasing for when $M > L$. This problem is not present for $M < L$, i.e. playing lower frequencies than 2^{-b} cycles/sample. Below this threshold, the maximum harmonics played back is limited by b.

Assuming that the waveform is bandlimited to B cycles/sample, we need to ensure that

$$\frac{M}{L}B \leq 0.5$$

as per the Nyquist criterion. Furthermore, B is determined by the number of harmonics h, i.e. $B = h2^{-b}$.

$$h2^{-b} \leq \frac{L}{M}0.5 \Leftrightarrow h \leq 2^{b-1}\frac{L}{M} = 2^{b-1}\frac{1}{\eta} \tag{3.12}$$

We can find the maximum number of harmonics (including the fundamental) h_{max} for a given digital frequency f_0. First, we find the required frequency scaling factor:

$$2^{-b}\eta = f_0 \Rightarrow \eta = f_0 2^b \Rightarrow \eta = N f_0 \tag{3.13}$$

Substituting equation 3.13 into equation 3.12, we find

$$h \leq \frac{0.5}{f} \Rightarrow h_{max} = \frac{0.5}{f_0}$$

If we use K buffers to store a varying number of harmonics, with the buffers indexed with $i \in \{0, 1, ..., K\}$, and store 2^{i+1} harmonics in each buffer, we are restricted to $K_{max} = b - 2$ as per equation 3.11. Note that this is arbitrary, and only to avoid large memory consumption for storing a linear increase in harmonics. If the MCU has external memory available, it should be considered to store as many waveforms with differing harmonics as possible, using a linear increase instead.

We can find the closes index of a given digital frequency f_0 which will contain the maximum number of harmonics (h_{max}), without aliasing, stored in the buffer i as follows:

$$i = \min(\lfloor log_2(\lfloor h_{max} \rfloor) \rfloor, K) \tag{3.14}$$

Equation 3.14 can be optimised by using a LUT and further memory considerations, to avoid an expensive call to a logarithmic function. See appendix B.4 for details. The C implementation for configuring a wavetable is shown in listing D.2 in the appendix.

Note that the minimum number of stored harmonics is 2 ($i = 0$). Thus we can only play frequencies up to a max of 0.25 cycles/sample. The highest note on an 88-key piano (C8) is 4186.01 Hz in equal temperament tuning. At a standard audio sampling rate of 44.1 kHz, this corresponds to $f_0 = 0.095 < 0.25$ cycles/sample. Almost all of the MIDI notes (see table C.2 in the appendix) are covered in this range, with the highest non-aliased note being E9 (10.54808 KHz), which corresponds to $f_0 = 0.239 < 0.25$ cycles/sample. Arguably, this is well outside the commonly played range. At 44.1 KHz, the only aliased MIDI notes are F9, F#9 and G9. This problem is avoided at a sampling rate of 48 kHz.

3.4.2. Implementing frequency scaling

Frequency scaling can be trivially implemented without any upsampling, downsampling or explicit filtering. The frequency scaling factor η (from equation 3.13) can be used to update a pointer into a LUT ($W[i]$) storing a periodic waveform.

For a table consisting of 2^b samples, the pointer will be considered as the phase $\phi[n]$ of the waveform. The phase must be updated using:

$$\phi[n] = \text{mod}(\phi[n-1] + \eta, 2^b) \tag{3.15}$$

Here, the frequency scaling factor η represents the amount that the table pointer must increase by, which is known as **stride**.

The modulus operation ensures that the phase pointer never falls outside of the range of indices of the table, effectively creating the periodicity required. However, we do not expect $f_0 > 0.5$, thus we can simplify 3.15 to:

$$\phi[n] = \begin{cases} \phi[n-1] + \eta, \ \phi[n-1] + \eta < 2^b \\ \phi[n-1] + \eta - 2^b, \ \phi[n-1] + \eta > 2^b \end{cases} \tag{3.16}$$

Which eliminates using an expensive floating-point modulus operation.

With $\phi[n] \in [0, 2^b)$ due to the modulus operation, the samples of $W[i]$ can be linearly interpolated to $L[n]$ using equation 3.3:

$$L[n] = \mathrm{lerp}(W[\lfloor\phi[n]\rfloor], W[\mathrm{mod}(\lfloor\phi[n]\rfloor + 1, 2^b)], \{\phi[n]\}) \tag{3.17}$$

Equation 3.3 is easy to implement in code, since the floor of a positive floating-point number can be determined through integer conversion. The fractional part of a floating-point can be determined using 3.5.

Using an if-statement to detect wrapping to the beginning of the table is also an alternative, but not necessarily as efficient, depending on branch penalties in the processor.

3.4.3. Implementing inter-wavetable interpolation

From subsection 3.3.2, the wavetables are stored in a 3D array, notated as $W[t,k,n]$, where $t \in \{0,1,2,3\}$ is the wave type, k is the harmonic index and n the sample index.

Interpolation between 2 wavetables can be specified by $\hat{t} \in [0,4)$, which is the continous extension of t. Interpolation can then be done similar to equation 3.17. Equation 3.18 shows this, also allowing wrapping to occur from the square to the sinusoid LUTs. Listing 3.3 shows the C implementation for wavetable sampling.

$$W(\hat{t},k,n) = \mathrm{lerp}(W[\lfloor\hat{t}\rfloor,k,n], W[\mathrm{mod}(\lfloor\hat{t}\rfloor + 1, 4),k,n], \{\hat{t}\}) \tag{3.18}$$

```c
inline float wt_sample(wavetable* wt, gen_config* gc) {
  uint8_t t = (uint8_t)gc->wt_pos;
  uint8_t tp1 = t + 1;
  np1 = fast_mod(np1, 4); //wrap LUT (square to sine)
  float x1 = lut_lookup(basic_luts[t][wt->harmonic_index], LUT_SIZE, wt->phase);
  float x2 = lut_lookup(basic_luts[tp1][wt->harmonic_index], LUT_SIZE, wt->phase);
  wt->phase += wt->stride;
  if (wt->phase > (float)LUT_SIZE) wt->phase -= (float)LUT_SIZE; //wrap phase
  return lerp(x1, x2, gc->wt_pos - (float)t);
}
```

Listing 3.3: Sampling from a wavetable

3.4.4. Applying FM

FM can be applied by modifying the the base stride for each sample. Similar to conventional FM techniques, we have a center/base frequency f_0, which is modified by by a modulation frequency $f_{FM}[n]$, with $\eta_0 = N f_0$ and $\eta_{FM}[n] = N f_{FM}[n]$ as per equation 3.13.

Thus, we store our base stride (η_0) as part of the internal wavetable state, which is set during a note-on trigger. We then have our final stride expressed as $\eta_f[n] = \eta_0 + \eta_{FM}[n]$.

3.5. IIR filtering

As detailed in section 2.2, filters are an essential component in many synthesisers. Filter cut-off can be controlled by external modulation sources. Thus, fast filter coefficient calculation is paramount, since it is calculated on a per-sample basis.

To achieve maximal speed, division operations must be avoided as far as possible, since most processors do not have single-cycle division operations. Furthermore, a biquad IIR filter can be used for speed. This also reflects the typical 2-pole filters often used in analogue synthesisers. The 1-pole filter will also be analysed, since it is often used for a “softer” roll-off and is required for anti-aliasing techniques in waveshaping, explored in section 3.6.

In this section, the 3 most common 2-pole filter types (HP24, BP12, LP24) along with 1-pole filters (HP12, LP12) for synthesis, will be detailed.

This section uses the techniques mentioned in section 2.4.3. Equations 3.19 to 3.23 are the prototype HP24, LP24, BP12, HP12 and LP12 filters respectively.

$$H_{hp24}(s) = \frac{s^2}{s^2 + \frac{1}{Q}s + 1} \tag{3.19}$$

$$H_{lp24}(s) = \frac{1}{s^2 + \frac{1}{Q}s + 1} \tag{3.20}$$

$$H_{bp12}(s) = \frac{s}{s^2 + \frac{1}{Q}s + 1} \tag{3.21}$$

$$H_{hp12}(s) = \frac{s}{s+1} \tag{3.22}$$

$$H_{lp12}(s) = \frac{1}{s+1} \tag{3.23}$$

The coefficients of the digital biquad filter can be calculated for the prototype filters. The denominator coefficients are the same for the HP24, LP24 and BP12 filters, whereas the denominator coefficients of the HP12 and LP12 are the same. It is summarised in table 3.2 below.

Table 3.2: Digital biquad filter denominator coefficients

Filter	$\mathbf{a_0}$	$\mathbf{a_1}$	$\mathbf{a_2}$
HP24, LP24, BP12	$2Q + \sin(\omega_0)$	$-2 \cdot 2Q\cos(\omega_0)$	$2Q - \sin(\omega_0)$
HP12, LP12	$(1-\cos(\omega_0)) + \sin(\omega_0)$	$2(1-\cos(\omega_0))$	$(1-\cos(\omega_0)) - \sin(\omega_0)$

The numerator coefficients are shown in table 3.3 below.

Table 3.3: Digital biquad filter numerator coefficients

Filter	$\mathbf{b_0}$	$\mathbf{b_1}$	$\mathbf{b_2}$
HP24	$-\frac{1}{2}b_1$	$-2Q(1+cos(\omega_0))$	$-\frac{1}{2}b_1$
LP24	$\frac{1}{2}b_1$	$2Q(1-cos(\omega_0))$	$\frac{1}{2}b_1$
BP12	$Qsin(\omega_0)$	0	$-b_0$
HP12	$-\sin(\omega_0)$	0	$-b_0$
LP12	$\frac{1}{2}b_1$	$2(1-\cos(\omega_0))$	$\frac{1}{2}b_1$

Calculating the coefficients can be optimised by using a LUT for cosine and sine approximation, and storing $2Q$, $\cos(\omega_0)$, $\sin(\omega_0)$, and $1-\cos(\omega_0)$ as intermediate values. Unfortunately a single division operation is unavoidable. All coefficients must be normalised by a_0 to obtain the filter form of equation 2.6. This scaling factor can also be stored as an intermediate value and then further used via multiplication.

Listing D.3 in the appendix shows the C implementation for calculating the LP24 filter coefficients using tables 3.2 and 3.3. Implementation for other filters are similar, utilising pre-negation of a_1 and a_2 to save on MCU clock cycles, storing intermediate values and utilising trigonometric LUTs, which are gained without extra memory consumption since a sinusoid is already stored as a basic waveform. The cosine LUT is also gained without extra memory consumption by using the identity $\cos(x) = sin(x + \frac{\pi}{2})$. The trigonometric lookup functions’ C implementations are shown in appendix listing D.5.

Filtering is then done using equation 2.7. The C implementation is shown in code listing D.4 in the appendix.

3.6. Waveshaper

As per the functional requirements, waveshaping will be done by one of 2 user-chosen functions: the hyperbolic tangent or the sinusoid. LUTs will be used to perform the waveshaping.

The sinusoidal LUT is already present in the basic waveforms, so no extra memory consumption is required. The hyperbolic tangent function will require its own LUT.

Shaping a discreet input function $x[n]$ to $y[n]$ with a continous waveshaping function $w(x)$ can be described with $y[n] = w(g \cdot x[n])$, where g is the gain into the waveshaping function. If w is an odd function, it will only add odd harmonics, whereas an even function only adds even harmonics. Both of the considered waveshaping functions are odd, so only odd harmonics will be added. The waveshaping function can be considered as a means to add signal distortion to our system.

Adding extra harmonics introduces aliasing into the system, since the waveshaping functions will add harmonics with amplitudes proportional to the input gain g. Aliasing can be reduced in a variety of ways, such as bandlimiting our signal into the waveshaping function, or by oversampling the input signal, or both. Even though oversampling is viable, it will require more processing. Thus, for simplicity and speed, only the bandlimiting option was considered. The system block diagram for this component is shown in figure 3.5.

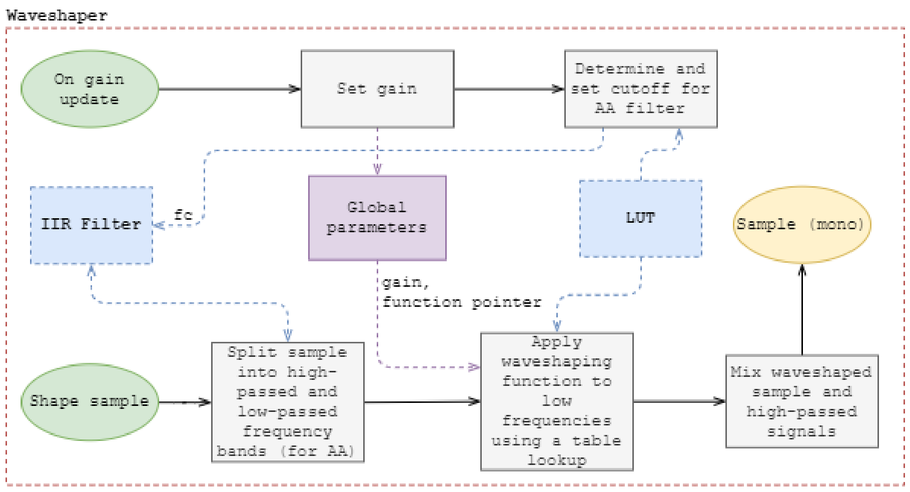

Waveshaper
On gain update
Set gain
Determine and set cutoff for AA filter
IIR Filter
fc
Global parameters
LUT
Sample (mono)
gain, function pointer
Shape sample
Split sample into high-passed and low-passed frequency bands (for AA)
Apply waveshaping function to low frequencies using a table lookup
Mix waveshaped sample and high-passed signals

Figure 3.8: Waveshaper system block diagram

3.6.1. Constructing the hyperbolic tangent LUT

To construct a buffer with $N+1$ samples indexed by $n \in \{0, 1, .., N\}$ storing the hyperbolic tangent, we need to consider some of the properties of $\tanh(x)$:

$$\lim_{x \to +\infty} \tanh(x) = 1 \text{ and } \lim_{x \to -\infty} \tanh(x) = -1 \tag{3.24}$$

$$\frac{d}{dx} \tanh(x) = \operatorname{sech}^2(x) \coth(x) \tag{3.25}$$

$$\lim_{x \to \pm\infty} \frac{d}{dx} \tanh(x) = 0 \tag{3.26}$$

We can approximate the behaviour of this function with $y[n]$ by storing a section of $\tanh(x)$ mapped to $x = \frac{n}{N} \in [0, 1)$, and then approximating the asymptotes in equation 3.24 by straight lines for $x > 1$ and $x < 0$. Shifting and scaling the function slightly is required, achieved by A (amplitude scaling) and k (input axis scaling). The function is also shifted right by $\frac{1}{2}$ to fit most of the function behaviour in [0,1).

$$y[n] = A\tanh(k(\frac{n}{N} - \frac{1}{2})) \tag{3.27}$$

Note that the gradient $y'[\frac{N}{2}] = k$. Thus, k is also a measure of the gradient steepness at $y[\frac{N}{2}] = 0$. We want $y[N] = 1$ and $y'[N] = \varepsilon$, where ε is the acceptable gradient error close to 0, from equations 3.25 and 3.26. Solving for these boundary conditions yields $A = \coth(\frac{1}{2}k)$, and values for k as a function of ε is shown in table 3.4.

Table 3.4: Axis scaling values and gradient errors for constructing the hyperbolic tangent LUT.

ε	**k**
0.1	5.3697
0.01	8.0810
0.001	10.6606
0.0001	13.1750

It should be noted that a higher k corresponds to better accuracy at the edges, but less accuracy within the curve around the x-intercept, since fewer sample points are available there for linear interpolation, due to axis scaling. To balance these factors, a value of $k = 9$ is chosen. Figure **??** in the appendix shows the LUT resulting from this section's design choices.

3.6.2. LUT indexing

Hyperbolic tangent

To index into the hyperbolic tangent LUT, we must reverse the transformations applied in equation 3.27. Thus our lookup index n for an input value t is shown by equation 3.28. We do not scale the final output by $\frac{1}{A}$, since $A \approx 1$ for small ε. For $x < 0$, we output -1 and for $x > 1$ we output +1.

$$n = \frac{N}{k}(x + \frac{1}{2})\ ,\ x \in (0, 1) \tag{3.28}$$

Sinusoid

We already have a LUT storing $\sin(2\pi\frac{n}{N})$ in the basic waveforms. Since we are interested in finding $\sin(x)$, we can find n utilising equation 3.6 (section 3.3), with $p = 2\pi$ and $A = N$ to ensure in-bounds indices for n and periodic behaviour.

$$n = N\{\frac{t}{2\pi}\} \tag{3.29}$$

3.6.3. Analysing waveshaping frequency content

Analytical analysis of the Fourier series of $\tanh(g \cdot sin(x))$ is possible by using the Taylor series expansion of $\tanh(x)$. However, the Taylor series only converges for $x \in (-\frac{\pi}{2}, +\frac{\pi}{2})$, which proves to be problematic for any input $|g \cdot x[n]| \geq \frac{\pi}{2}$. Furthermore, the Fourier series of $\sin(g \cdot sin(x))$ can be evaluated in terms of the Bessel J functions, which would require a LUT anyways, since these functions have an infinite series representation.

For any unknown input $g \cdot f(x)$ into the waveshaping function, finding an analytical solution of the Fourier series is incredibly cumbersome, or impossible. Instead, a numerical approach is taken to determine the effects of waveshaping in the frequency domain, where the effect of g will be investigated on a sinusoid. We will devise our own technique.

For any waveshaping function $w(x)$ and a set of M increasing gain values $\{g_1, g_2, ..., g_M\}$, we can investigate the effect of g in the discreet time domain, by sampling a 1 Hz sinusoid with N points within a period, while recording P periods.

$$Y[k,g] = \mathrm{DTFT}\{w(g \cdot \sin[2\pi \frac{n}{N}])\} \tag{3.30}$$

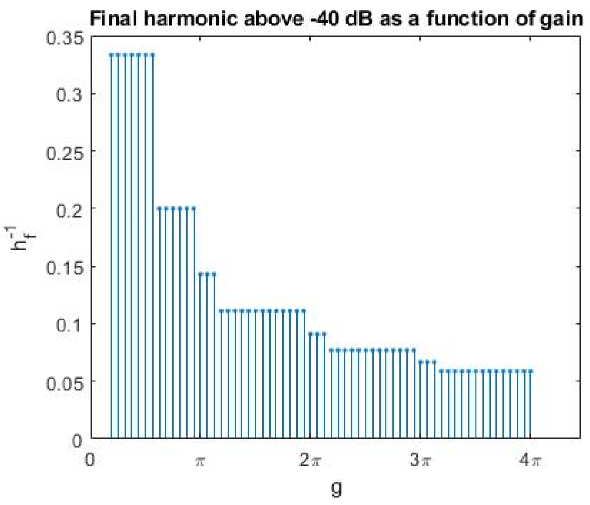
For each $g \in \{g_1, g_2, ..., g_M\}$, we can record the number of the harmonic that is last in exceeding a predefined amplitude ratio with the fundamental. In this case, our ratio is chosen as $\frac{1}{100}$ (-40 dB) to the fundamental.

We must ensure that we choose N large enough, so that we can avoid significant aliasing in our analysis that could contaminate the results, and we choose P large enough for good frequency domain resolution. Choosing $N = 1000$ (500 times more than the Nyquist limit) and $P = 100$ proved to be adequate for the set of gains that were analysed. From this choice, the fundamental occurs at $k = 100$, with harmonics occurring at $k = 100 \cdot h$, $h \in \mathbb{N}/\{1\}$. Up to 500 harmonics may be analysed. The hyperbolic tangent had $g \in \{1, 2, ..., 64\}$ analysed. The sinusoid had gains analysed for $g \in \{\frac{\pi}{16}, \frac{2\pi}{16}, \frac{3\pi}{16}, ..., 4\pi\}$. This limits the gains that the user can apply to 64 and 4π respectively.

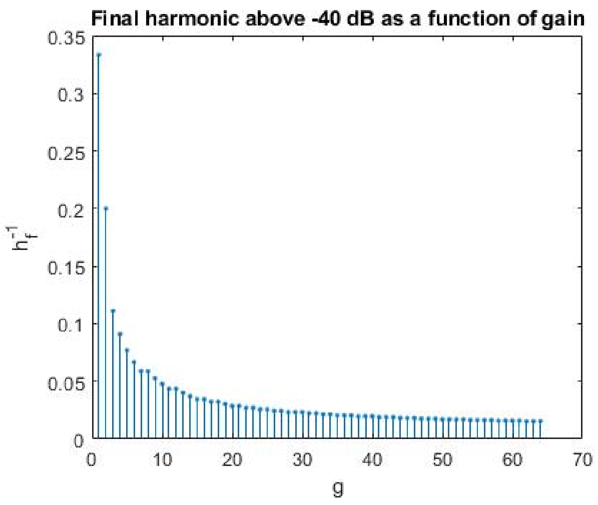
Knowing the last harmonic h_f above the -40 dB to the fundamental threshold, we can determine the bandwidth B_w in cycles per sample using the Nyquist criterion, shown in equation 3.31.

$$B_w = \frac{0.5}{h_f} \tag{3.31}$$

Figure 3.9 plots h_f^{-1} as a function of g for both waveshaping functions. These values can be used in a LUT to determine the cutoff frequency for a bandwidth-limiting LPF. For $g = 0$ and $h_f^{-1} = \infty$ (seen in figure 3.9a for $g \in \{\frac{\pi}{16}, \frac{2\pi}{16}\}$), we can set $h_f^{-1} = 2\frac{20000}{f_s}$, which limits the signal to the audible range as per equation 3.31.



(a) Sinusoidal waveshaping harmonic analysis.



(b) Hyperbolic tangent waveshaping harmonic analysis.

Figure 3.9: Waveshaping harmonic analyses.

3.6.4. Anti-aliasing filters

From the previous section, we can now determine the maximum bandwidth of an input as a function of gain. However, by limiting the bandwidth, we discard the frequency content above the bandwidth. This is undesirable. Hence, we must both high-pass and low-pass our incoming signal, and combine them again after waveshaping has occurred to the low-passed signal.

We have already designed IIR filters in section 3.5. We can choose between LP12 and LP24 filters. Although a LP24 filter would suppress aliasing better, the LP12 filter is chosen. This is so that we can gain more harmonic content, with aliasing as a trade-off, which is often preferable in musical circumstances. This choice is made with the end-product audio in mind.

The LP12 filter also has a nice property, that could remove the need for a corresponding HP12 filter entirely. From equation 3.23 and 3.22, we can run them in parallel.

$$H_{||}(s) = H_{lp12}(s) + H_{hp12}(s) = \frac{1}{s+1} + \frac{s}{s+1} = 1 \tag{3.32}$$

From , we can then construct the HP12 filter from the LP12-filtered signal as shown in equation 3.33. This eliminates the need for a another filter, and replaces it with a subtraction operation.

$$H_{hp12}(s) = 1 - H_{lp12}(s) \tag{3.33}$$

We can express this in the discreet-time damain as follows:

$$y_{hp12}[n] = x[n] - h_{lp12}[n] * x[n] \tag{3.34}$$

Given a waveshaping-function $w(x)$, we can write the entire waveshaping system with input $x[n]$ and output $y_w[n]$ as shown in equation 3.35, using equations 3.33 and 3.34. We can store the low-passed signal $x_{lp12}[n] = h_{lp12}[n] * x[n]$ as an intermediary value to save clock cycles. We therefore only require a single IIR filter for this component.

$$y_w[n] = x[n] - h_{lp12}[n] * x[n] + w(h_{lp12}[n] * x[n]) \tag{3.35}$$

Equation 3.35 is the concrete form of what is depicted in the system block diagram from figure 3.8. The convolution operator represents the discreet-time difference equation (2.7) from section 2.4.3.

3.7. ADSR envelope generator

The ADSR envelope generator can be considered a state machine that produces a piecewise-defined function of 3 exponentials and a constant, based on a trigger signal, as shown in figure **??**. Figure 3.10 shows the block diagram of this component. This is a state-machine with 5 states: attack; decay; sustain; release; not playing. The attack and release states can only be externally triggered, therefore, the sampling mechanism only needs to be concerned with transitions to the decay, release and not playing states. A “playing” state is any state except the “not playing” state.

For code simplicity, states are encoded using integers: $\{0,1,2,3,4\}$, referring to attack, decay, sustain, release and not playing respectively. Furthermore, the attack, decay and release times must have a minimum value to avoid clicks and pops in the audio. This value was chosen as 1 ms.

3.7.1. Creating the exponential LUT

To construct the LUT, we consider an upwards-decaying exponential $E[n]$, with $E[0] = 0$ and $E[N-1] = 1$, where N is our required number of samples. Equation 3.36 shows the form we are interested in.

$$E[n] = K(1 - R^n),\ R \in [0,1) \tag{3.36}$$

Since $\lim_{n\to\infty} E[n] = K$, we choose a threshold value $p \in (0,1)$ for our final sample, such that $E[N-1] = pK$. An analogue circuit would usually use a comparator with such a threshold voltage to charge and discharge a

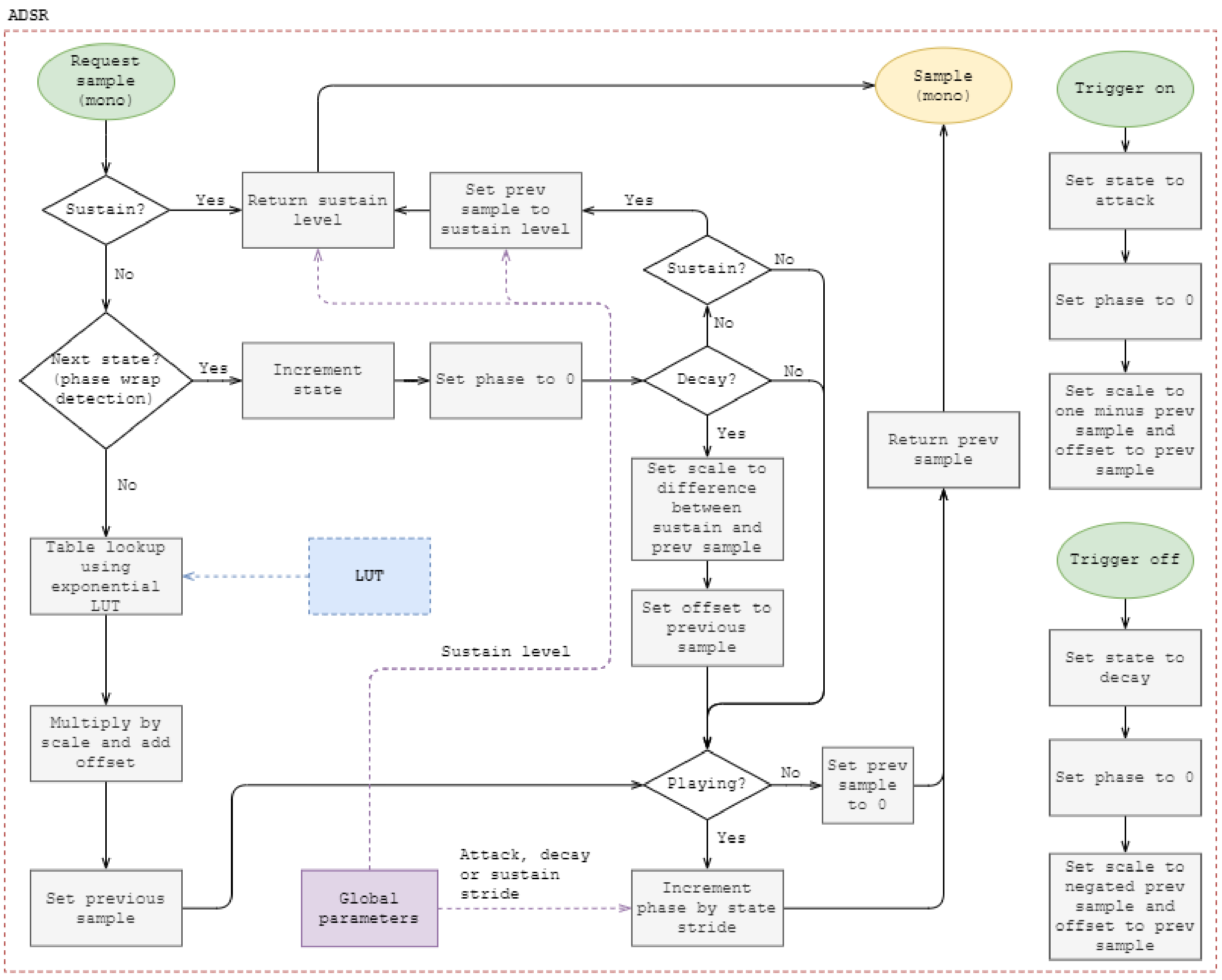

ADSR
Request sample (mono)
Sustain?
Yes
No
Return sustain level
Set prev sample to sustain level
Yes
Sustain?
No
No
Sample (mono)
Trigger on
Set state to attack
Set phase to 0
Set scale to one minus prev sample and offset to prev sample
Next state? (phase wrap detection)
Yes
Increment state
Set phase to 0
Decay?
No
Yes
Set scale to difference between sustain and prev sample
Set offset to previous sample
Return prev sample
No
Table lookup using exponential LUT
LUT
Sustain level
Multiply by scale and add offset
Playing?
No
Set prev sample to 0
Yes
Trigger off
Set state to decay
Set phase to 0
Set scale to negated prev sample and offset to prev sample
Set previous sample
Global parameters
Attack, decay or sustain stride
Increment phase by state stride

Figure 3.10: ADSR envelope system block diagram.

capacitor, which also yields an exponential function. Such a threshold is arbitrary, and usually at the designer's discretion. It will determine the shape of the function stored in the LUT. A higher p will result in a flatter slope as $n \to N-1$. A typical value to choose is $p = \frac{2}{3}$. Substituting our boundary conditions into 3.36, we find:

$$K = \frac{1}{p}$$

$$R = \sqrt[N-1]{1-p}$$

3.7.2. Implementing the state-machine

The nature of this component requires many state checks to function properly. Since this can cause branch penalties, care must be taken when considering the amount of branches and their locations, by minimising branching if possible.

Proper re-trigger behaviour is required, as per the specifications. Thus, the ADSR requires storage of its most recent sample, to allow for scaling and offset calculation based off previous output. It cannot be known in advance when triggering will occur, which implies that a release state trigger can occur during any state. This is similar for the attack state. A data structure is required to store all the internal state of a single ADSR envelope. The internal state includes LUT phase, exponential scaling and offset values, the previous sample, and the current state.

As per subsection 3.7.1, we know that the last value of the LUT is 1, and has a range of $[0,1]$. The previous value ($y[n-1]$) is also recorded as part of internal state. We need to calculate the offset (b) and the scaling factor (a) such that we can get the required behaviour from $E[n]$. We can deduce the following:

1. A trigger on event must initiate the attack phase, which must rise to 1. Thus, $a = 1 - y[n-1]$ and $c = y[n-1]$.
2. A trigger off event must initiate the release phase, which must decay to 0. We then have $a = -y[n-1]$ and $c = y[n-1]$.
3. The decay phase must decay to the sustain level (s). We cannot be certain that it transitions from a sample that is exactly 1 (due to linear interpolation and phase wrapping), so thus $a = s - y[n-1]$ and $c = y[n-1]$.

To achieve the correct attack, decay and release timings, we can treat the exponential LUT like wavetable, and using stride values that correspond with the required timing. However, we do not want it to exhibit periodic behaviour, so we must ensure that the phase is always less than $N-1$. A transition and phase reset must occur after the phase exceeds $N-1$. To calculate the required stride η to sweep over a single exponential ADSR state, given a time T_0 in seconds and a fixed LUT size N, we derive an expression from equation 3.13:

$$\eta = \frac{N}{T_0 f_s} \tag{3.37}$$

The C implementation of the state machine is shown in listing D.6 in the appendix.

3.8. Generator

The generator is a top-level component responsible for managing all sub-components and creating the stereo audio samples for a single note. It receives note on/off triggers that correctly configures all sub-components with the required parameters and subsequently triggers the ADSR envelopes. Figure 3.11 shows the system block diagram of this component. This component also requires internal state, which is not shown in the block diagram to prevent clutter.

The internal state of the generator, which is a function of the configured frequency, must be configured with a note-on trigger. The details of this configuration is discussed further in this section.

3.8.1. Note-on/off triggers

With reference to the system block diagram (figure 3.11), the triggering process will be discussed. It can never be known when a note trigger will take place, or at which frequency it takes place. Thus, wavetable frequencies for the center, left and right channels must be configured using the MIDI note ID (see table C.2) which can be used as a lookup index for the required digital frequency. The note frequency, reffered to as **base frequency**, is stored as part of the internal state. Although not required, the phase of the wavetables are reset.

A note on/off trigger must also trigger the volume and filter ADSR envelope components on/off.

3.8.2. Vibrato

Vibrato can be applied using the FM capabilities of the wavetable component from section 3.4.4. From the global parameters, the vibrato intensity v is specified as a value in cents, which can be used to calculate the required frequency deviation amplitude A as a function of the base frequency f_0.

$$A = 2^{\frac{v}{1200}} f_0 - f_0 = (2^{\frac{v}{1200}} - 1) f_0 \tag{3.38}$$

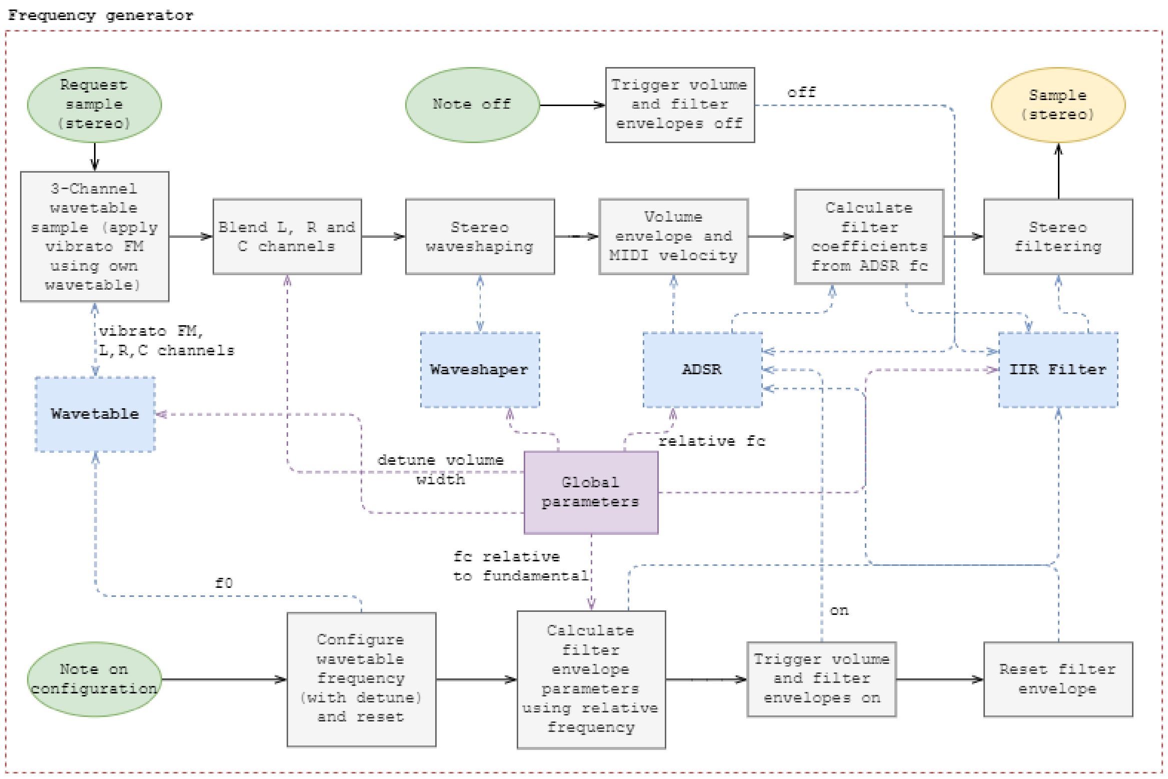

Frequency generator
Request sample (stereo)
Note off
Trigger volume and filter envelopes off
off
Sample (stereo)
3-Channel wavetable sample (apply vibrato FM using own wavetable)
Blend L, R and C channels
Stereo waveshaping
Volume envelope and MIDI velocity
Calculate filter coefficients from ADSR fc
Stereo filtering
vibrato FM, L,R,C channels
Waveshaper
ADSR
IIR Filter
Wavetable
detune volume width
Global parameters
relative fc
fc relative to fundamental
f0
on
Note on configuration
Configure wavetable frequency (with detune) and reset
Calculate filter envelope parameters using relative frequency
Trigger volume and filter envelopes on
Reset filter envelope

Figure 3.11: System block diagram of the generator.

A sinusoidal wavetable with the required vibrato frequency f_v can used to create the frequency deviation samples $df[n]$. The FM must be applied to the center, left and right channels.

From equation 3.38, the frequency scaling vibrato factor $2^{\frac{v}{1200}}$ can be stored as part of the internal state of the generator, which can be efficiently calculated using the techniques from subsection 2.4.1. Equation 3.39 shows the frequency deviation as a result of vibrato. Note that the frequency deviation is expected to be small ($<$ 50 cents), so no recalculation of the harmonic index of the wavetables will be done. On edge cases, this could cause aliasing, but would be unnoticeable.

$$df[n] = A\sin(2\pi f_v n) \tag{3.39}$$

The C implementation is shown in code listing D.7 in the appendix.

3.8.3. Filter cutoff modulation

The filter cutoff frequency $f_c[n]$ in samples/cycle is relative to the digital frequency of the note that is playing (f_0), as the functional requirements. A relative starting and ending cutoff frequency specified by scaling factors r_s and r_e are used to determine amount of cutoff modulation ($m_{filter}[n]$) applied by the filter ADSR envelope. Care must be taken to ensure that the filter cut-off does note exceed the Nyquist limit. To ensure that the cut-off always remains within reasonable limits, it is restricted to the audible range (20 Hz to 20 kHz).

$$f_c[n] = \max(\min(r_s f_0 + f_0(r_e - r_s)m_{filter}[n], \frac{20000}{f_s}), \frac{20}{f_s}) \tag{3.40}$$

From equation 3.40, we can store $r_s f_0$ and $f_0(r_e - r_s)$ in the internal generator state as filter modulation offset and amplitude respectively. This will save some clock cycles when sampling.

3.8.4. Volume modulation

The note volume $v[n] \in [0,1]$ is determined by the ADSR envelope modulation $m_{vol}[n]$ and the MIDI note velocity $k_{vel} \in \{0,1,...,127\}$, which is a 7-bit unsigned integer that is specified with a note-on MIDI message [35]. Since $m_{vol}[n] \in [0,1]$ by design, we need to normalise k_{vel}.

$$v[n] = \frac{k_{vel} m_{vol}[n]}{127} \Rightarrow V[n] \in [0,1] \tag{3.41}$$

From equation 3.41, we can save on clock cycles by storing $\frac{k_{vel}}{127}$ in the internal generator state with a note-on configuration.

3.8.5. Stereo width

There are a variety of ways and functions that can be used to introduce stereo width, given center ($y_C[n]$), left ($y_L[n]$) and right ($y_R[n]$) channels that need to be mixed into stereo audio.

From subsection 2.4.2, we can use the mid-side form of audio to specify the mono and stereo content of the audio. As per the requirements, the detune width ($\delta_w \in [0,1]$) and detune volume ($\delta_v \in [0,1]$) parameters must be used to blend the detuned left and right channels. Volume is a scaling factor, where a width of $\delta_w = 0$ corresponds to no stereo width, and $\delta_w = 1$ to full stereo width. Using these definitions we can define the mid-side content as follows:

$$M = y_C[n] + \frac{1}{2}\delta_v(2-\delta_w)(y_R[n] + y_L[n]) \tag{3.42}$$

$$S = \frac{1}{2}\delta_v \delta_w (y_L - y_R) \tag{3.43}$$

Note that when $\delta_w = 1$, the mid channel contains the absolute minimum of $0.5\delta_v$ of the detuned content. This is necessary to ensure that the “mono’d” signal will not result in a complete loss of detune content. The center channel is only contained in the mid information, with the difference between left and right channels (see equation 2.3) is scaled by the detune width and the volume.

Substituting equations 3.42 and 3.43 into equations 2.4 and 2.5, we determine the content of the stereo audio to be as follows:

$$L = y_C + \delta_v(y_L + (1-\delta_w)y_R)$$

$$R = y_C + \delta_v(y_R + (1-\delta_w)y_L)$$

The C implementation for sampling from a generator is shown in code listing D.8 in the appendix.

3.9. Generator manager

The generator manager is responsible for triggering inactive generator components. This system stores an array of a finite number of generators, which need to be assigned to notes as appropriate. As per the functional specification, if all generators are active, then the oldest active generator must be re-triggered and configured to play any new incoming notes. The expected amount that re-triggering will be performed depends on user parameters. A generator is only considered as “inactive” or “available” if the release state of the volume envelope has finished. Otherwise it is “active” .

The amount of samples generated will be far greater than the amount of note-on/off requests. However, it is still necessary to make generators available when they exited their volume envelope release phase, which must be done once for every buffer request. The smaller the buffer, the less the latency (see equation 3.1), but the higher the generator polling rate. Thus, speed is of importance here. It cannot be known when a generator will be available until the release phase is over. The release time could also change, if the user changes this parameter while an envelope is in that state. The active generators must be polled at the start of every buffer request to check whether it can be made available.

The system block diagram for this component in shown in figure 3.12.

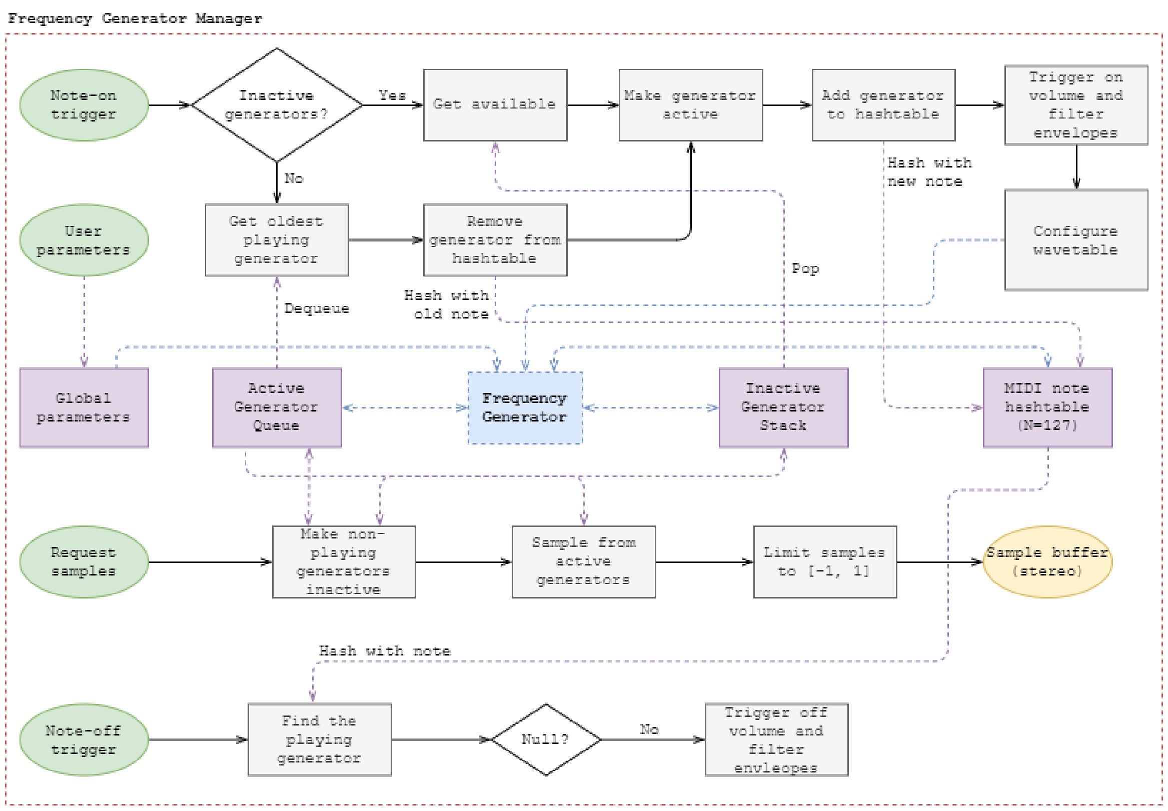


Figure 3.12: Generator manager system block diagram

MCUs often only have a finite amount of memory available to them. We therefore need to be certain about the memory consumption of the generators. For speed, storing them in an array (and not any other structures such as a C++ vector or queue) is best.

Creating and destroying generator data for every note is an unnecessary process that requires memory allocation, whereas allocating a fixed amount of memory for generators, that are stored at successive memory locations is preferable.

Pointers to the generator data can be used to add generators to other data constructs such as queues and stacks for management. Without calling any memory allocation operations (like “malloc”), we can effectively store the generator data at fixed locations within the heap. The may also help with caching.

See the appendix for the full C implementation in code listing D.9.

3.9.1. Implementing basic data structures with arrays

Efficient management of the generators requires the use of 3 data structures:

1. **Queue** - the FIFO nature of this structure is required to keep track of the order in which generators we triggered. The first item in this queue will be the oldest. It also keeps tracks of all the active generators, so that inactive generators are not sampled as well.

2. **Stack** - the stack has an efficient array implementation which allows for quick access to inactive/available generators.

3. **Hash table** - Note on and note off triggers come in pairs. It is necessary to keep track of which generator is playing which note, so that it can easily be triggered off. Since the MIDI protocol only supports 128 notes, a hash table provides a quick way to access that generator in $O(1)$ time complexity, with little memory consumption (512 bytes, for a 32-bit system).

With this management scheme, a generator can either be in the active queue, or the available stack, but not in both.

Suppose that our system contains N generators. The queue and the stack can easily be implemented with an array of size N, storing generator pointers. Another variable is required that points to the location of first available slot in the array - the "head pointer". Inserting into array, for both the queue and the stack, is done by inserting the item into the head pointer location, and then incrementing the pointer by 1.

Removing an item from the stack, requires the decrementing of the head pointer by 1, and then retrieving the data at that location. This operation has $O(1)$ time complexity. Removing an item from the queue is more expensive, since the data at location 0 must retrieved. The head pointer is decremented, and all items are shifted one to the left. This operation has $O(N)$ time complexity. In both cases, we never loop past the item before the head pointer, thus deleting items is not required.

Note that a more efficient queue implementation is possible ($O(1)$ time complexity), by having a starting and ending index, and treating the array as a circular buffer. But since N will never realistically be very large (≤ 32), and dequeuing items will not happen often, it adds unnecessary complexity by adding in extra indexing operations.

The hash table is simple to implement, using an array of size 128, which stores generator pointers. The MIDI note value can be used to index into the array. Adding and retrieving items have $O(1)$ time complexity.

3.9.2. Note-on/off triggers

With reference to figure 3.12, there are two cases to consider for a note-on trigger: triggering if inactive generators are present; re-triggering the oldest generator if all generators are active. To check whether any active generators are available is simple, if the head pointer of the inactive generator stack is at index 0, we can be sure that all generators are inactive.

If there are available generators, a generator is popped off the stack. It is associated with the note in the hashtable, and appropriately configured with the correct frequency. This requires two $0(1)$ operations. If no generators are available, then the oldest generator is dequeued, and removed from its associated note in the hashtable which is stored in its internal state. This is required so that the note-off trigger for the previous playing note will be ignored. Triggering, configuring and hashing proceeds as per the previous case. This process requires an $0(1)$ and an $O(N)$ operation. If a note-on trigger occurs for the same note before a note-off (which should not happen unless it is artificially forced or through faulty MIDI), then retrieving the generator from the hashtable is done. If nothing is present, only then may a new generator be assigned to a note, otherwise the old generator is re-triggered. This is required, since it could lock a generator out from ever being triggered off. This is shown in the C implementation, although not explicitly shown in the block diagram.

For note-off triggers, the MIDI note is used for a hashtable lookup. If there is a generator present, it is still associated with that note and is triggered off. Otherwise, the generator has been re-triggered, and nothing is done.

3.9.3. Sample buffer request

Once a sample buffer (of a fixed size M) is requested, all the active generators must be sampled. To maximise the use of caching, a single generator generates the required M samples before moving on to sampling from the next generator.

This is added to the stereo buffer, and scaled according to the number of voices. If we assume that V voices are managed by this component, we can approximate the output of each voice to be between -1 and +1 (high Q filtering and the detuned wavetables might make this range larger). We therefore scale the output of each voice by N^{-1}, to ensure that the output is roughly within the -1 and +1 range. Once all the generators are sampled, the buffer is hard-clipped to [-1, +1], so that we can be certain that no bit overflow will occur when converting the buffer data into an appropriate format for the codex IC. This conversion is hardware and application dependent (such as 24-bit PCM audio samples), and is not within scope.

Within this buffer request, we must free all inactive generators within the active queue, so that they may be used again. If we want to achieve this in $O(N)$ time, we can use a simple algorithm on the active generator queue, demonstrated in figure 3.13. In the figure, 2 buffer requests are shown, with the algorithm being execute twice.

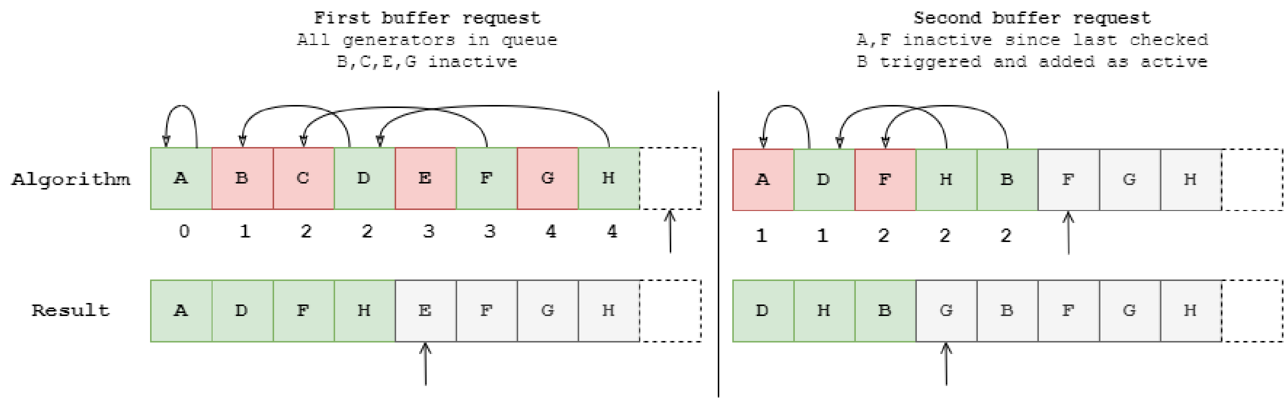


Figure 3.13: Generator queue freeing algorithm example

The generators are labeled from A-H, with available generators shown in red and active generators shown in green. The example manages 8 generators, with the dotted box at the end indicating a non-element location, which the head pointer can be pointed at if the queue is full. The grey elements are items that will never be accessed, since they are past, or at, the head pointer. The algorithm keeps a running count of the number of available generators. If the generator is available, it is pushed onto the available stack and removed from the hash table. Otherwise, if the generator is still active, it is shifted left by the number of counted available generators. The running count is shown below the generators in the figure.

After the algorithm has executed, the head pointer is decremented by the running count. The C implementation of this algorithm ("gm_make_not_playing_available") is seen in listing D.9 in the appendix.

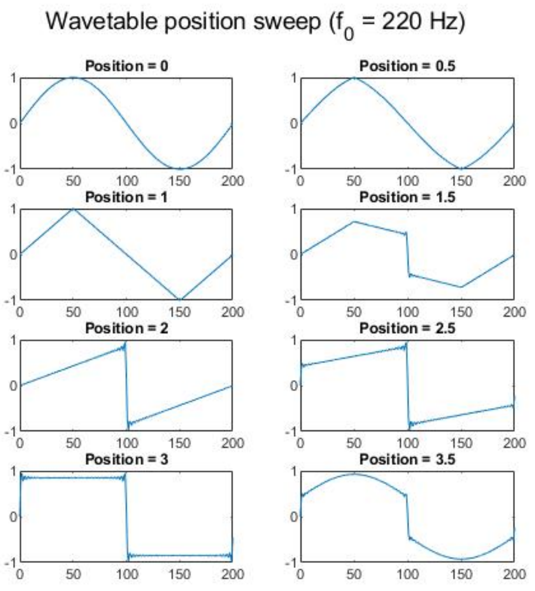
Chapter 4
System testing

Each component of the system is individually tested, along with a final complete system test, and a musical test with some qualitative comments, which is described in appendix B.6.

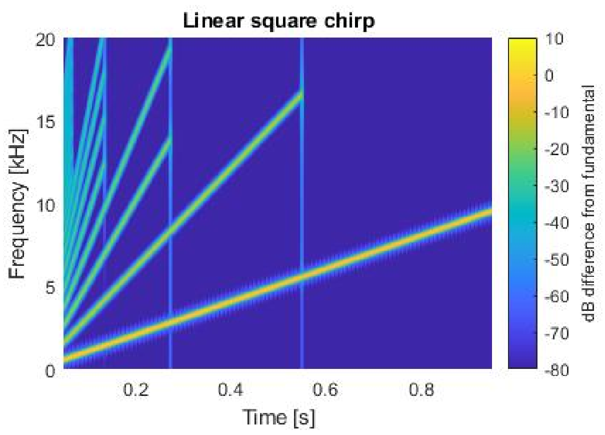
Unless otherwise mentioned, all spectrograms were generated with a window size of 512, a sample overlap of 510 for good time-axis resolution, using a Blackman windowing function for the greatest side-lobe attenuation, but large main-lobe width. This windowing function was chosen so that side-lobes do not contaminate the results, which was deemed more important than frequency accuracy. All tests were done at a sampling rate of 44.1 kHz, with 1 second's worth of samples (44100 samples), with wavetable LUT sizes of 512 samples and a maximum of 8 generators. All frequency sweeps (chirps) were done from 20 Hz to 10 kHz, since notes above 10 kHz are rarely played. The data was generated using C code and the implemented components (see appendix E for test code), and analysed using MATLAB.

4.1. Wavetable

A single wavetable set at a frequency of 220 Hz (A3) is sampled with different wavetable positions. This test is done to verify the inter-wavetable interpolation function. Figure 4.1a shows the results. Additional frequencies (55 Hz, 880 Hz) were also tested to show the effect of harmonic indexing, and are shown in figure A.4 in the appendix. To test the harmonic indexing detailed in section 3.4, a linear square-wave chirp was sampled from a single wavetable component, with frequency being updated once per sample. A spectrogram was then generated. This is shown in figure 4.1b.



(a) Wavetable position sweep for A3 (220 Hz).



(b) Linear 20 Hz - 10 KHz square-wave chirp (position = 3).

Figure 4.1: Testing inter-wavetable interpolation and harmonic indexing.

The inter-wavetable interpolation works as expected. Harmonic indexing also proves effective against aliasing, with no frequency producing aliased harmonics. The vertical lines in the spectrogram that is observed when harmonic loss occurs is result of the previous harmonics instantaneously disappearing, and can be neglected. Also note that the square wave only has uneven harmonics, but has a reduction in harmonics if the even harmonics will alias, which is why the harmonics do not sweep all the way up to the Nyquist frequency (22.05 kHz).

4.2. ADSR

An ADSR object was sampled an triggered at different points in time. The envelope had time-parameters set for $t_a = 0.125$s, $t_d = 0.125$s, $t_r = 0.125$s with a sustain of 0.5. The exponential LUT is constructed with $p = \frac{2}{3}$. Since 44100 samples were obtained, the triggering times were defined in terms of $k = \lfloor \frac{44100}{8} \rfloor$. A trigger-on is done at sample index $n_0 = \lfloor \frac{k}{3} \rfloor$ ($t \approx 0.042$s). A retrigger is done at $n_1 = \lfloor n_0 + k + \frac{k}{2} \rfloor$ ($t \approx 0.229$s). A trigger off is done at $n_2 = \lfloor n_1 + 3k \rfloor$ ($t \approx 0.604$s). Figure 4.2 shows the results.

The envelope operates at expected, with the correct timing, sustain level and range. The envelope was successfully retriggered.

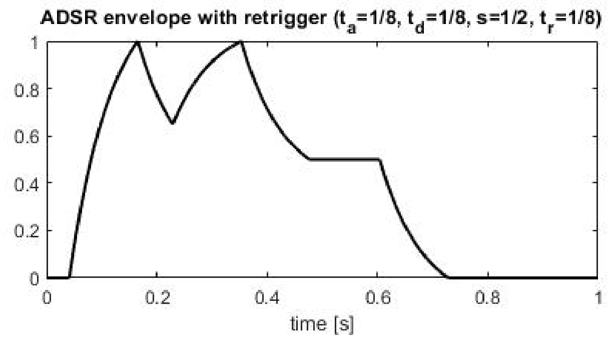
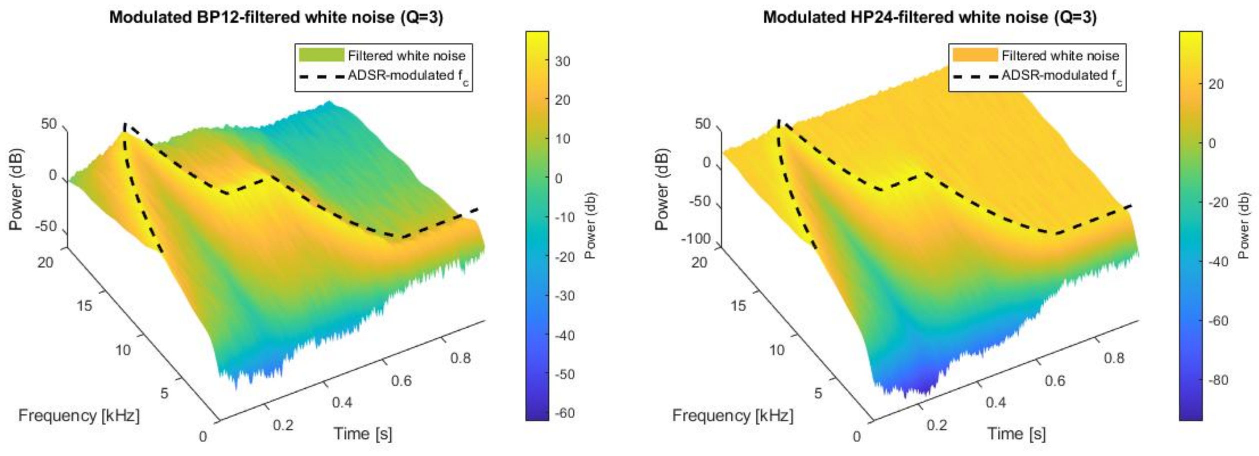


Figure 4.2: Sampled ADSR envelope with retrigger.

4.3. IIR filters

The time-dependent cutoff frequency of these filters were tested, which would simultaneously show correct filter operation. It should be noted that pin-point accurate cutoff frequencies are not tested, since deviation of a few Hertz due to LUT interpolation accuracy is inconsequential in a musical context. It is only the correct variation over time that is critical.



(a) BP12-filtered Guassian noise. **(b)** HP24-filtered Guassian noise.

Figure 4.3: Filtered white noise indicating the effect of time-varying filters.

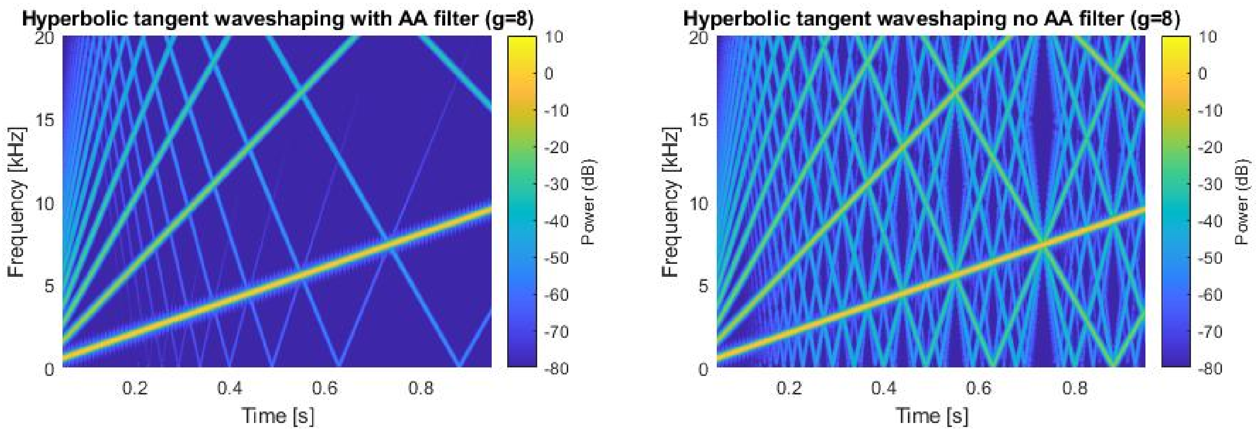
Guassian white noise ($\mu = 0$, $\sigma^2 = 1$) was generated to test filter operation. The white noise was filtered and a spectrogram was produced, shown in figure 4.3 for the HP24 and BP12 filters. The spectrograms of the other filters (LP24, LP12, HP12) is shown in figure A.5 in the appendix. The white noise contains frequency content at all frequencies, but unequal amounts at any instantaneous point within a STFT. An ADSR envelope was used to modulate the signal, with $t_a = 0.25$s, $t_d = 0.1$s, $t_r = 0.2$s and a sustain of 0.5. The signal was modulated to have a minimum frequency of 1 kHz and a maximum of 20 kHz. The BP12, HP24 and LP24 filters had a Q set to 3, to make the cutoff frequency more apparent. To make the results more interpretable, a 2D Guassion blur was applied to the time-frequency STFT data, using a Guassian smoothing kernel with $\sigma = 15$. This eliminates the jagged edges of the white noise's STFT, but does not change the filter's shape significantly

The envelope's output was also separately recorded, and plotted as a dotted line for reference. The spectrogram has a window size of 1024 with a sample overlap of 1020, allowing for high time and frequency resolution.

The figures clearly show a correct cutoff frequency modulation, with the required range between 1 kHz and 20 kHz. It should be noted that it does not explicitly start at 1 kHz, since the 1024 samples required to perform the first FFT consumes the start of the envelope.

4.4. Waveshaping

Linear sinuoidal chirps (20 Hz - 10 kHz) generated by a generator component with detune volume set to 0, position set to 0, no filtering applied other than for anti-aliasing, and ADSR envelope times set to the minimum of 1 ms with a sustain of 1. The envelope only affects the signal in the attack phase, which occurs at the start of the chirp. This effect is insignificant over the time-span of the recorded data.

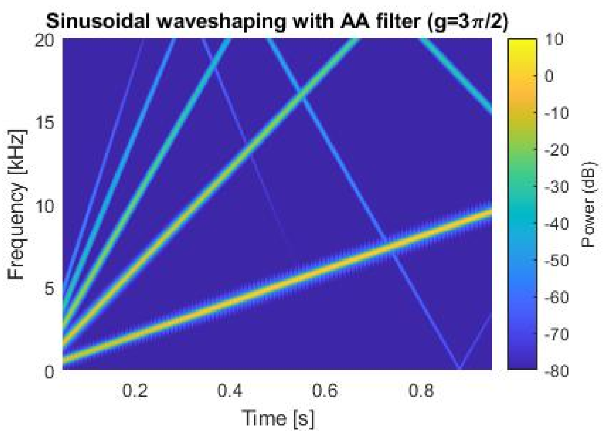


(a) Waveshaped chirp for $g = 8$, with anti-aliasing filter. **(b)** Waveshaped chirp for $g = 8$, without the anti-aliasing filter.

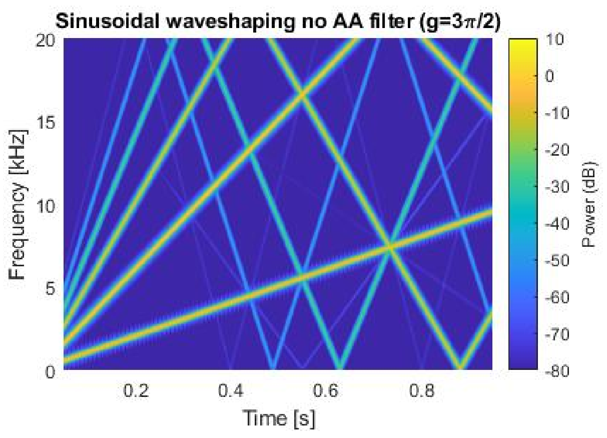
Figure 4.4: Comparing the effect of the anti-aliasing LP12 filter for hyperbolic tangent waveshaping.

The gain into both the sinusoidal and the hyperbolic tangent waveshaping functions with changed, with the anti-aliasing LP12 filter's cut-off set to the appropriate value from the look-up table as described in section 3.6. A spectrogram was generated, and compared to the results from having the anti-aliasing filter set to a cutoff of 20 kHz. The comparison is shown in figures 4.4 and 4.6. The output of the STFT was normalised by the maximum value in the time-frequency data, so that the effect of the harmonics relative to the fundamental can more easily be seen.

The reduction in aliasing for the hyperbolic tangent is clear. It is also apparent that the magnitude of the aliased frequencies in figure 4.4a is smaller than that of 4.4b. Similar observations are made for figure 4.5a and 4.5b, although the extra harmonic content added by sinusoidal waveshaping is less for these examples.



(a) Waveshaped chirp for $g = \frac{3}{2}\pi$, with anti-aliasing filter.



(b) Waveshaped chirp for $g = \frac{3}{2}\pi$, without the anti-aliasing filter.

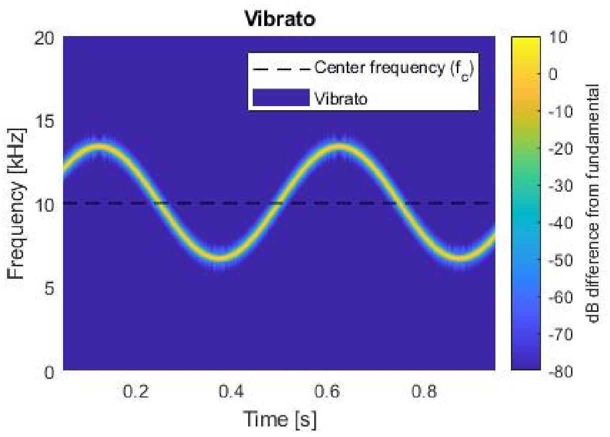
Figure 4.5: Comparing the effect of the anti-aliasing LP12 filter for sinusoidal waveshaping.

Additional gains were also tested, and are shown in figures A.6 to A.9 in the appendix.

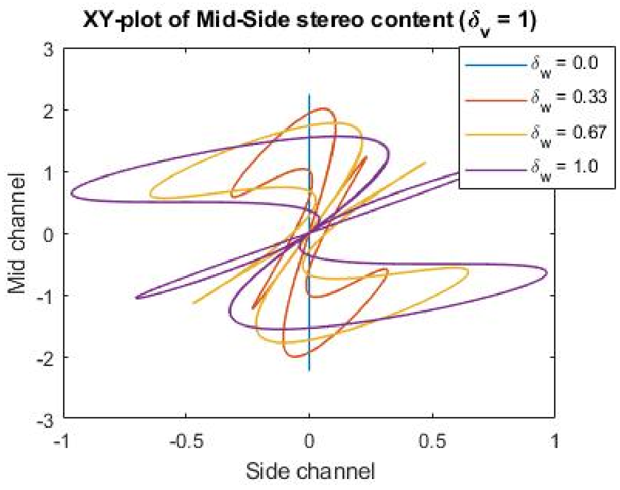
4.5. Generator

The vibrato and stereo capabilities of the generator component were tested in this section. All relevant generator parameters were set so that their effect will non-existing or minimal (detune volume set to 0; ADSR envelope times set to 1 ms, with a 1.0 sustain value; no filtering or waveshaping). A sinusoid with a base frequency of 10 kHz, frequency deviation (vibrato intensity) of 500 cents ($10 \cdot 2^{500/1200} - 10) = 3.35$ kHz deviation), and FM frequency of 2 Hz was sampled. A spectrogram was created with the sampled data normalised by the maximum value in the time-frequency data, along with a straight line that shows the expected center frequency of 10 kHz. This is shown in figure 4.6a.

To test the stereo width, the detune volume is changed to 1, and various stereo widths sampled. The additional left and right oscillators were detuned up and down by 1200 cents (1 octave) respectively. A sinusoid was used, with no filtering applied. The base frequency was arbitrarily set to 100 Hz, which does not matter for the following test.



(a) Sinusoidal vibrato.



(b) Lissajous plot of the mid and side stereo content.

Figure 4.6: Testing vibrato and stereo width.

The left and right audio channels were converted to mid and side format as per equations 2.2 and 2.3. An XY-plot (common known as a Lissajous plot in the music industry) was created from these channels, where the

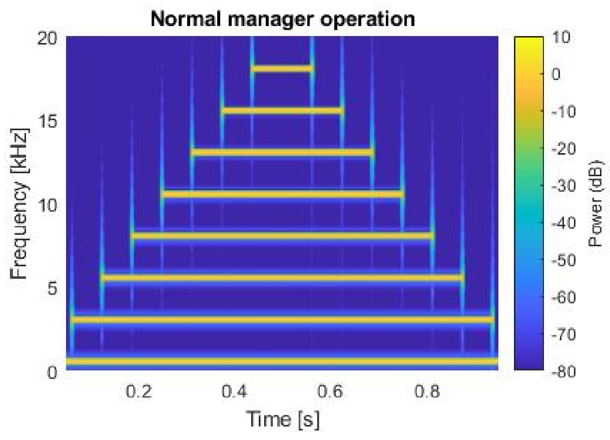
Y-component represents the mid channel and the X-component represents the side channel. For a mono signal ($L = R$), a straight vertical line must be observed. For a phase inverted signal ($L = -R$), a straight horisontal line must be observed. The results are shown in figure 4.6b.

Figure 4.6a shows that the sinusoidal vibrato acts as expected. Figure 4.6b clearly indicates an increase in side-channel content as the stereo width increases, which can be seen by the axis of symmetry tending towards the side-channel axis.

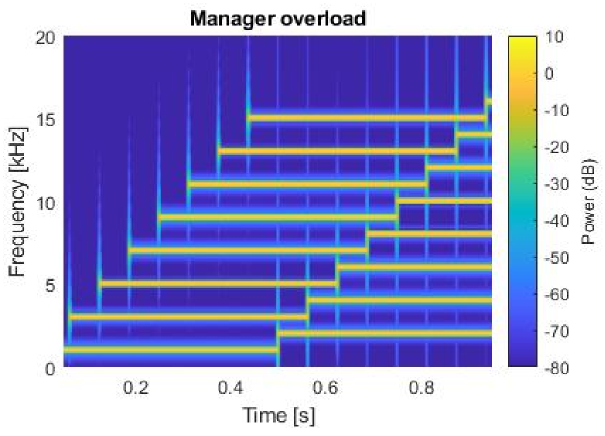
4.6. Generator manager

Normal operation of the generator manager was tested. This includes note on/off triggers, below capacity. A set of 8 predefined frequencies $\{0.5, 3, ..., 8\}$ kHz was associated with the MIDI notes $\{0, 1, ..., 7\}$. Each sinusoidal tone is triggered on at $t = \{0, 1/16, ..., 7/16\}$ and triggered off in reverse order at $t = \{8/16, 9/16, ..., 15/16\}$. Detune volume is set to 0, ADSR time parameters to 1 ms with a sustain of 1, and no filtering or waveshaping is applied. The results are shown in figure 4.7a.

Generator overload is tested, with frequencies $\{1, 3, ..., 15, 2, 4, ..., 16\}$ kHz associated with MIDI notes $\{0, 1, ..., 15\}$. Each sinusoidal tone is sequentially triggered at $t = \{0, 1/16, ..., 15/16\}$, with no off-triggers occurring. This is shown in figure 4.7b.



(a) Testing normal manager operation.



(b) Testing the manager when overloaded.

Figure 4.7: Testing the generator manager.

For both results, the STFT is normalised by the maximum within the time-frequency data. Also note that the spectral lines at the edges of note on and off triggers are a result of the sudden addition of the tones, and are not of concern.

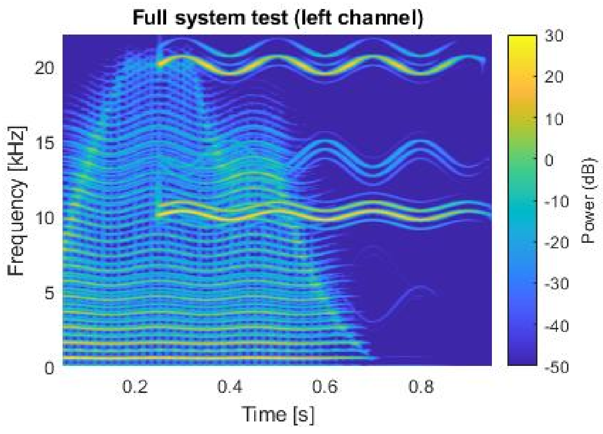
The generator manager operates as required by the specifications, under both normal and overload conditions.

4.7. Full system test

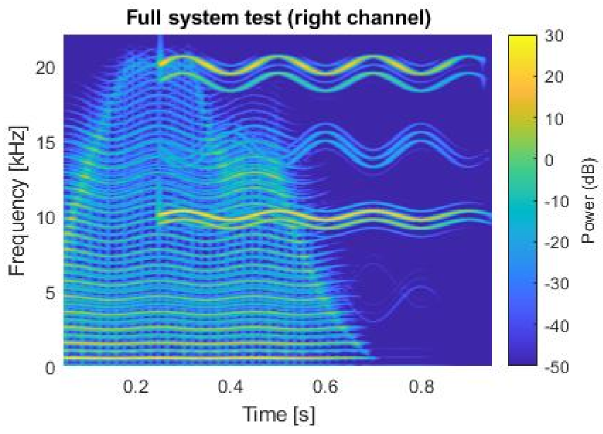
The entire system is tested by triggering the generator manager with 2 notes, configured with frequencies of 500 Hz and 10 kHz, and associated with MIDI notes 0 and 1 respectively. MIDI note 0 was triggered on at $n = 0$ and off at $n = 22048$ ($t \approx 0.5$ s). MIDI note 1 was triggered on at $n = 11024$ ($t \approx 0.25$ s) and off at $n = 44096$ ($t \approx 1.0$ s). Due to the complexity of this system, the exact outcomes and their measurements are difficult to predict. Instead, we investigate whether correct operation is present through inspection.

The parameters were chosen so that their effect can be clearly seen in the time domain and spectrogram data. The wavetables were configured with a position of 2.5, so that both odd and even harmonics are present, a detune of 100 cents (approximately 600 Hz deviation at 10 kHz), detune volume of 0.25 and a detune width of 1, so that the left and right channels show a clear detune distinction. A LP24 filter with a Q of 7, relative cut-off frequency start of 1 and end of 50 were chosen, such that the filter envelope can be observed due to high Q and sweeps an observable range for MIDI note 0. The filter envelope has an attack of 300 ms, decay of 100 ms, sustain of 0.5 and release of 200 ms. A hyperbolic tangent waveshaper with a gain of 2 is used, along with vibrato with an intensity of 50 cents and frequency of 5 Hz. The volume envelope envelope has an attack of 10 ms, decay of 200 ms, sustain of 0.5 and release of 200 ms.

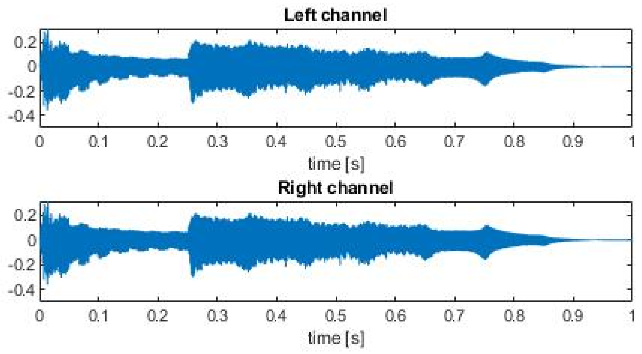
Spectrograms were generated for the left and right audio channels, using a window size of 1024 with a 1020 sample overlap. This is shown in figure 4.8a and 4.8b. The audio is also plotted to demonstrate the effect of the volume envelope, shown in figure 4.8c.



(a) Left channel spectrogram.



(b) Right channel spectrogram.



(c) Left and right channel audio.

Figure 4.8: Full system test ($f_1 = 500$ Hz, $f_2 = 10$ kHz).

The spectrograms show correct filter envelope modulation, which is capped to 20 kHz. Vibrato with the correct frequency deviation can can clearly be observed, with some aliasing present due to waveshaping. Figure 4.8a shows the additional detuned oscillator to be higher in frequency, with reduced amplitude from the the fundamental, whereas figure 4.8b shows the additional detuned oscillator to be lower in frequency. Only a single detuned oscillator is present per audio channel, as is expected for a width of 1.

Figure 4.8c adequately demonstrates the effect of the volume envelope, but is obscured due to the high-Q filter sweep. The addition of the second generator can be clearly be observed at 0.25 seconds.

Please see appendix B.6 for a full musical test, with some qualitative comments on the produced audio. The appendix also provides links where the audio can be listened to.

Chapter 5
Summary and Conclusion

5.1. Results achieved

The audio generation core for a hardware synthesiser was successfully designed and implemented in C. The following, with reference to subsection 1.3.2, was successfully achieved:

1. The basic waveforms (sine, triangle, sawtooth and square) can be selected and interpolated according to user parameters, by using the wavetable component. The LUTs required for this task has been effectively designed to combat aliasing, while allowing for an arbitrary amount of copies with increasing harmonics. This system supports any sampling frequency, and can yield a bit depth of up to 24 bits if required.
2. Two additional oscillators can be detuned from the base frequency oscillator, and panned so as to achieve a desired stereo width, using the aforementioned wavetable techniques.
3. An ADSR envelope state machine has been designed that can produce the required piecewise-exponential behaviour, with retriggering capabilities.
4. HP12, HP24, LP12, LP24 and BP24 filters derived from analogue prototypes have been implemented, requiring only 2 table lookups and a single division operation to calculate the FIR coefficients.
5. The cutoff frequency of the available filters can be modulated by an ADSR envelope on a per-sample basis, and set relative to the fundamental frequency.
6. Stereo waveshaping using the hyperbolic tangent and sinusoid functions can be performed, with an effective anti-aliasing method being implemented, using a single LP12 filter, with a cutoff frequency set as a function of the gain into the waveshaper. Waveshaping is done efficiently using LUTs.
7. Sinusoidal vibrato at a specified frequency and intensity can be applied to a stereo waveform using the FM capabilities of the wavetable component.
8. Polyphony is achieved using a manager component that can play up to a fixed number of notes, which can do so efficiently in O(N) time. If a new note is triggered when the generator is at capacity, the oldest playing note is retriggered. The manager can produce stereo samples from active generators, and store it into a buffer of arbitrary size, while provide adequate headroom (samples are within [-1,1] for normal operation, and clipped if necessary).

All the components designed in this thesis are building blocks for creating a more complex synthesis core. The building blocks can easily be rearranged in the signal chain to produce different effects, and can be used on their own for other applications. See appendix B.5 for detail on these alternatives and applications.

5.2. Further improvements and work

Designing a wavetable synthesiser can be a highly creative and product-dependent process, with unique features often being the selling point. This thesis only explored and designed the fundamental aspects of what can become fully-fledged product. The first step in furthering design would be the implementation of the designed software on hardware. An ARM-based implementation is recommended, such as the STM32 Cortex M7 series. A dual-core processor could be desirable, by using the lower-speed core to process user input and handle display, while using the high-speed core exclusively for DSP. Implementation on a Cortex A7-series MPUs can also be desirable if greater performance is required. Simulating and inspecting the ARM assembly could also be done, where various optimisation techniques can be applied to improve performance. A performance analysis can also be done for specific hardware architectures, and optimised accordingly.

Furthermore, the designed software can also be implemented for VST design, where modifications can be made for concurrency, by sampling generator components on different threads. This could also be beneficial for some Cortex A7 MPUs. The techniques detailed in this thesis could be also be implemented on an FPGA-based design.

It should be noted that fully-fledged MIDI support is not integrated into this system. Allowing pitch-bend and hold-pedal functionality is considered critical for products. Pitch-bend and hold-pedal functionality would be a possible augmentation to this system, without requiring significant effort. A MIDI-control based system can also be added, that would allow the MIDI protocol to also change user parameters, so that the instrument can be externally controlled.

Aside from hardware implementation, many additions can be made to the designed components. Among these additions are the implementation of more filters, such as the comb, or peak filter. The filters can also have more “character” added, by utilising saturation in the feedback path. This can be achieved through including hyperbolic tangent waveshaping in the filtering function, which then saturates the output $y[n]$. This would incur further numerical simulation to avoid aliasing for feedback saturation. Furthermore, the addition of oversampling in combination with the described anti-aliasing waveshaping scheme can be introduced for more robust performance. The optimal thresholds to choose for harmonics in the proposed scheme (section 3.6) can also be revisited by doing an in-depth study of acceptable levels of aliasing for musical contexts. The design of the ADSR state-machine can also be revisited for optisation.

The proposed harmonic indexing scheme can also be revisited, where a more memory-efficient way of storing LUTs with the required number of harmonics can be devised. This could perhaps be achieved by storing a single waveform with many harmonics, and then filtering and temporarily storing it to reduce harmonic content when required. This requires many design considerations, determined by the effect of linear interpolation in the frequency domain discussed in section 3.17. This would increase processing requirements, and would be best achieved using a second core, or a multi-threaded system. Such a technique would be hardware dependent, which is why it was not considered in this thesis, which aimed to provide a general foundation for these systems. The current technique only requires an external memory for LUT storage, which can be set up to contain many megabytes of waveforms, loaded by the MCU as required.

Finally, a system can be designed that would allow for arbitrary parameter modulation, by having a selection of modulator functions that the user can assign to modulate arbitrary paramaters. This would increase computational and implementation complexity significantly.

Finally, integrating other effects, such as a general EQ, delay, phaser and reverb effects, could be beneficial for the release of a finished product. Such effects would require a powerful processor if it is combined with this system, or it can be offloaded to an additional effects processor.

Bibliography

[1] *ADSR A-140*, Doepfer. [Online]. Available: https://doepfer.de/a100_man/A140_man.pdf

[2] *Configure the Coefficients for Digital Biquad Filters in TLV320AIC3xxx Family*, Texas Instruments, 2010. [Online]. Available: https://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/6/Configure-the-Coefficients-for-Digital-Biquad-Filters-in-TLV320AIc3xxx-F_2E00__2E00__2E00_.pdf

[3] MTU Physics, "Tuning: Frequencies for equal-tempered scale, A4 = 440 Hz," last accessed 20 July 2021. [Online]. Available: https://pages.mtu.edu/~suits/notefreqs.html

[4] ARM Developer, "FPU instruction set," last accessed 9 October 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0439/b/BEHJADED

[5] Seveen, "STM32 Synth," 2019, last accessed 16 October 2021. [Online]. Available: https://github.com/Seveen/stm32-synth

[6] audiojs, "List of common audio sample rates," 2017, last accessed 14 October 2021. [Online]. Available: https://github.com/audiojs/sample-rate

[7] Maxim Integrated , "Audio Data Converters - Parametric Search," 2021, last accessed 1 October 2021. [Online]. Available: https://www.maximintegrated.com/en/products/parametric/search.html?fam=audiocodecs&295=OR%7CAudio%20CODEC

[8] S. Lee, "This is the early history of the synthesizer," 2018, last accessed 2 October 2021. [Online]. Available: https://www.redbull.com/gb-en/electronic-music-early-history-of-the-synth

[9] AJH Synth, "Transistor ladder filter," last accessed 2 October 2021. [Online]. Available: https://ajhsynth.com/VCF.html

[10] Make Noise, "Maths," last accessed 2 October 2021. [Online]. Available: https://www.makenoisemusic.com/modules/maths

[11] MusicRadar, "The 16 best eurorack modules 2021: the right modules for any build, or expansion of your modular synthesizer system," 2021, last accessed 2 October 2021. [Online]. Available: https://www.musicradar.com/news/the-best-eurorack-modules-in-the-world

[12] Vintage Synth Explorer, "Yamaha DX7," last accessed 2 October 2021. [Online]. Available: https://www.vintagesynth.com/yamaha/dx7.php

[13] Xfer, "Serum: Advanced wavetable synthesizer," last accessed 3 October 2021. [Online]. Available: https://xferrecords.com/products/serum

[14] Native Instruments, "Massive," last accessed 3 October 2021. [Online]. Available: https://www.native-instruments.com/en/products/komplete/synths/massive/

[15] Arturia, "Pigments: Polychrome software synthesizer," last accessed 4 October 2021. [Online]. Available: https://www.arturia.com/products/analog-classics/pigments/overview#en

[16] D. Karras, "Sound synthesis theory," 2018, last accessed 3 October 2021. [Online]. Available: https://en.wikibooks.org/wiki/Sound_Synthesis_Theory

[17] Detroit Modular, "Eurorack: Modules," last accessed 7 October 2021. [Online]. Available: https://www.detroitmodular.com/eurorack.html

[18] Ableton, "Live," last accessed 1 October 2021. [Online]. Available: https://www.ableton.com/en/shop/live/

[19] Spectrasonics, "Omnisphere 2.8 - endless possibilities," last accessed 5 October 2021. [Online]. Available: https://www.spectrasonics.net/products/omnisphere/

[20] P. Mantione, "The fundamentals of physical modeling synthesis," 2019, last accessed 5 October 2021. [Online]. Available: https://theproaudiofiles.com/physical-modeling-synthesis/

[21] J. Smith, "Virtual acoustic musical instruments: Review and update," *Journal of New Music Research*, vol. 33, pp. 283–304, 09 2004.

[22] Native Instruments, "Kontakt 6 player," last accessed 5 October 2021. [Online]. Available: https://www.native-instruments.com/en/products/komplete/samplers/kontakt-6-player/

[23] Spitfire Audio, "Spitfire audio," last accessed 5 October 2021. [Online]. Available: https://www.spitfireaudio.com/shop/

[24] M. Apler, "Sound industry history: East vs. west coast synthesis: Moog, buchla, and finding a balance between familiarity and uncertainty in electronic instrument design," last accessed 7 October 2021. [Online]. Available: https://hii-mag.com/allposts/industry-historyeastwest

[25] Doepfer, "A-111-3 Micro Precision VCO / VCLFO," last accessed 7 October 2021. [Online]. Available: https://doepfer.de/A1113.htm

[26] Nord, "Nord Stage 3," last accessed 7 October 2021. [Online]. Available: https://www.nordkeyboards.com/products/nord-stage-3

[27] IntelliJel, "The little filter that could," last accessed 7 October 2021. [Online]. Available: https://intellijel.com/shop/eurorack/uvcf/

[28] *VCA Module*, MFB. [Online]. Available: http://mfberlin.de/wp-content/uploads/VCA_english.pdf

[29] S. Pigeon, "The non-linearities of the human ear," last accessed 7 October 2021. [Online]. Available: https://www.audiocheck.net/soundtests_nonlinear.php

[30] HyperPhysics, "Equal temperament," last accessed 7 October 2021. [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/Music/et.html

[31] H. Robjohns, "Q. How does Mid-Sides encoding/decoding actually work?" 2017, last accessed 8 July 2021. [Online]. Available: https://www.soundonsound.com/sound-advice/q-how-does-mid-sides-recording-actually-work

[32] W. Abu-Al-Saud, "Differential pulse code modulation (dpcm)," last accessed 8 October 2021. [Online]. Available: https://faculty.kfupm.edu.sa/EE/wajih/files/EE%20370/EE%20370,%20Lecture%2025.pdf

[33] Wikimedia Foundation, Inc., "Audio bit depth," 2021, last accessed 8 October 2021. [Online]. Available: https://en.wikipedia.org/wiki/Audio_bit_depth

[34] J. O. Smith, *Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications, Second Edition*, last accessed 30 July 2021. [Online]. Available: http://ccrma.stanford.edu/~jos/mdft/

[35] MIDI Association, "Summary of MIDI 1.0 Messages," 2021, last accessed 9 October 2021. [Online]. Available: https://www.midi.org/specifications-old/item/table-1-summary-of-midi-message

[36] Wikimedia Foundation, Inc., "Q (number format)," 2021, last accessed 10 October 2021. [Online]. Available: https://en.wikipedia.org/wiki/Q_(number_format)

[37] M. Bhojasia, "Explain Logical and Arithmetic Shifts in C Language with Examples," last accessed 10 October 2021. [Online]. Available: https://www.sanfoundry.com/c-tutorials-logical-arithmetic-shifts-uses/

Appendix A

Additional figures

(a) Doepfer A-111-3

(b) IntelliJel UVCF

(c) MFB VCA

(d) Doepfer A-140

Figure A.1: Examples of fundamental modules.

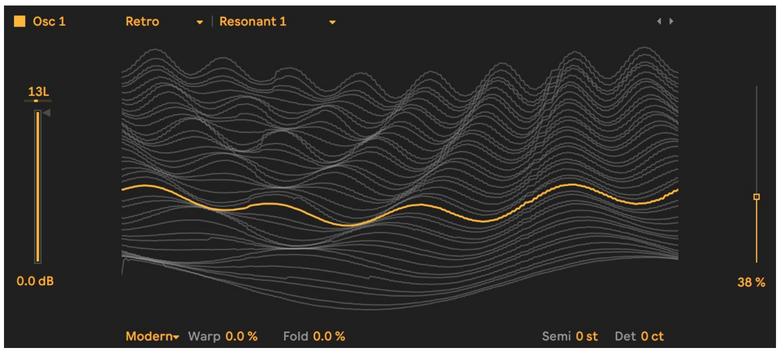
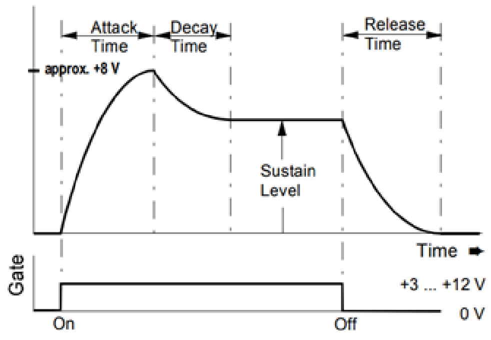
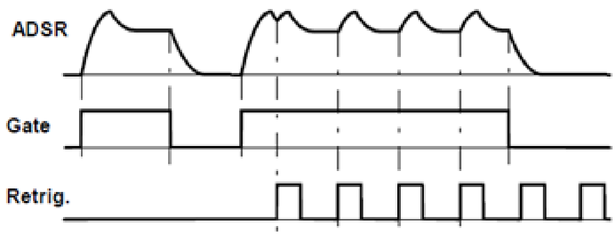


Figure A.2: Ableton's Wavetable VST

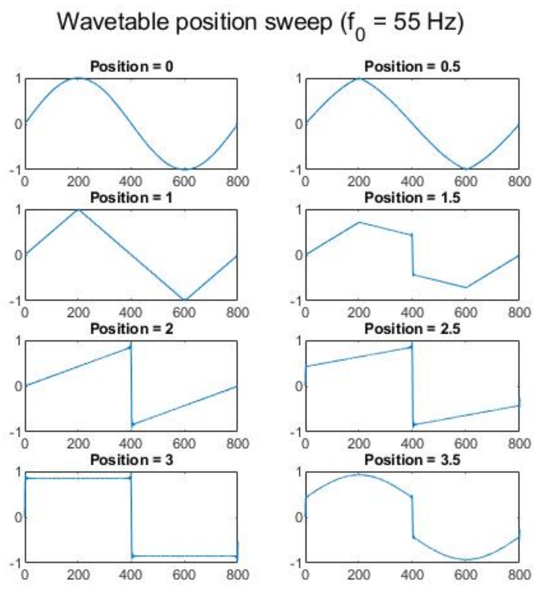


(a) A typical ADSR trigger with a gate signal.

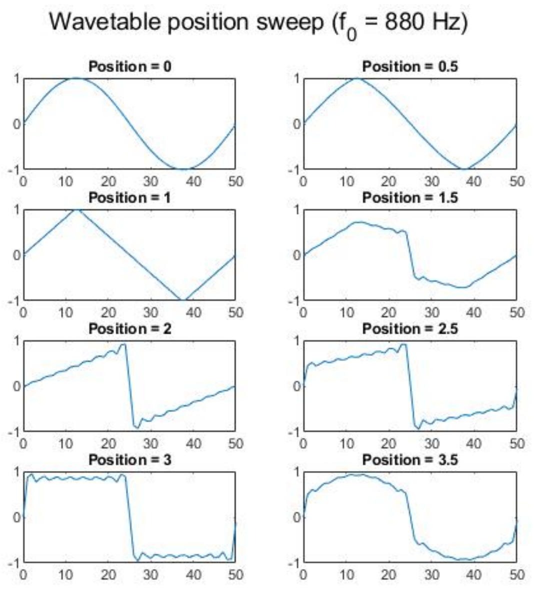


(b) A typical ADSR re-trigger.

Figure A.3: Doepfer A-140 ADSR operation from [1].

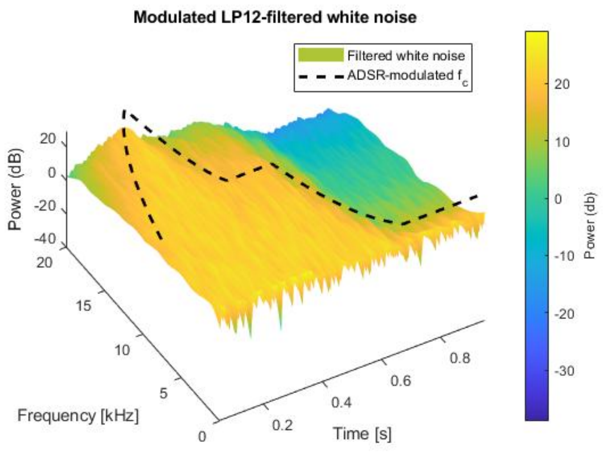


(a) Wavetable position sweep for A1 (55 Hz).

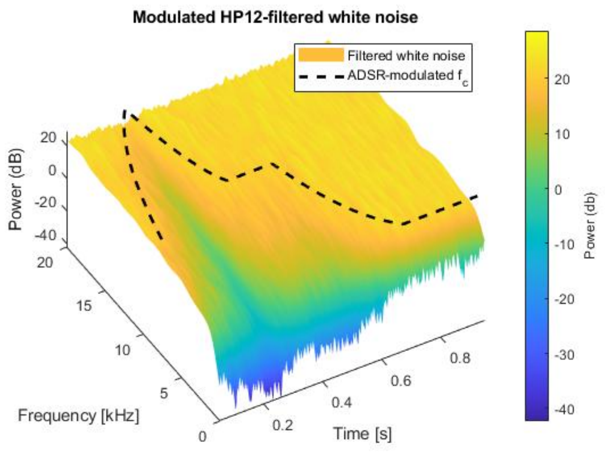


(b) Wavetable position sweep for A5 (880 Hz).

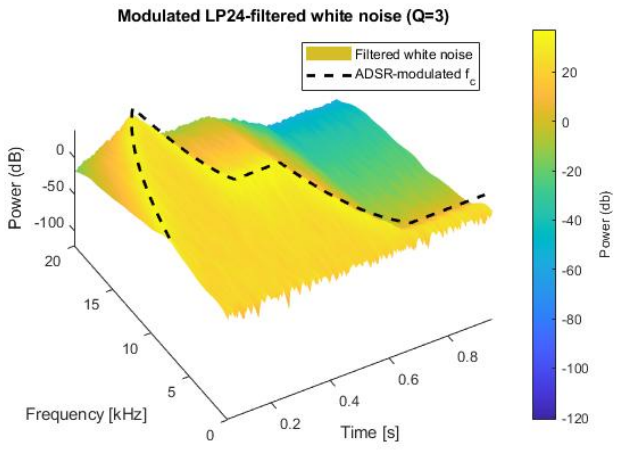
Figure A.4: Testing inter-wavetable interpolation at different frequencies.



(a) LP12-filtered Guassian noise.

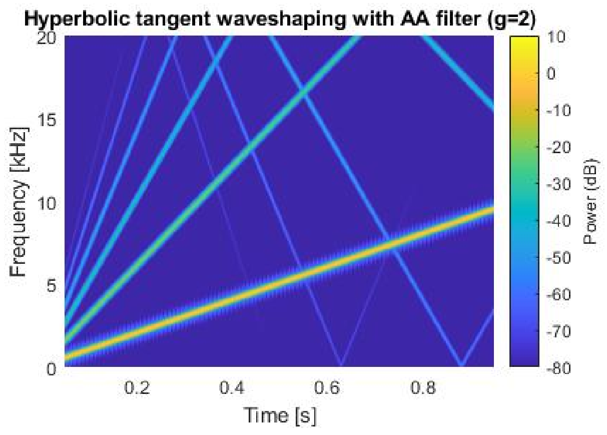


(b) HP12-filtered Guassian noise.

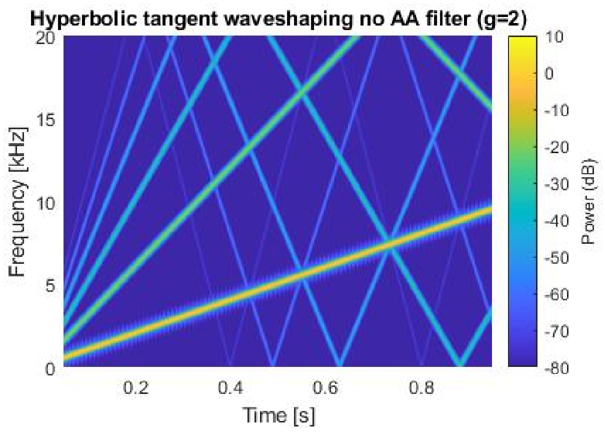


(c) LP24-filtered Guassian noise.

Figure A.5: Filtered white noise indicating the effect of time-varying filters.

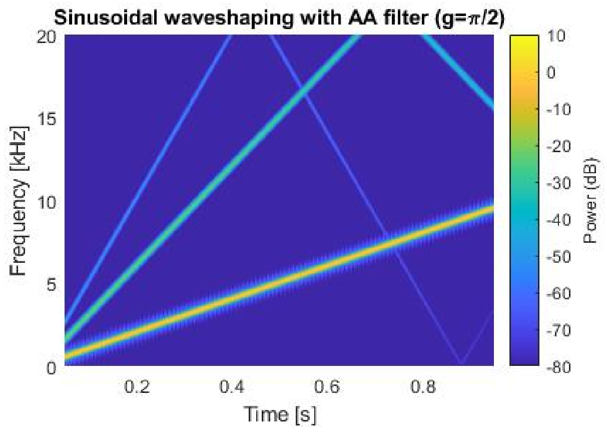


(a) Waveshaped chirp for $g = 2$, with anti-aliasing filter.

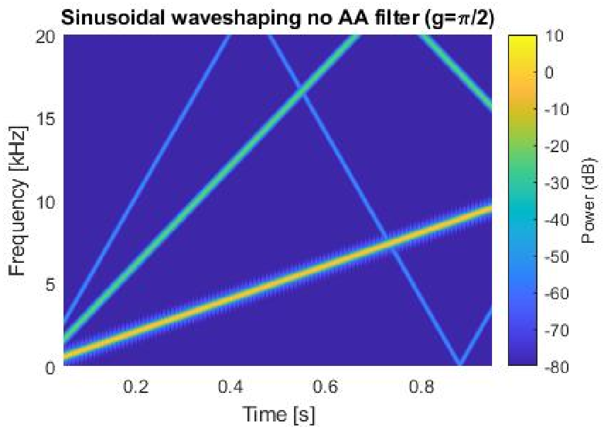


(b) Waveshaped chirp for $g = 2$, without the anti-aliasing filter.

Figure A.6: Comparing the effect of the anti-aliasing LP12 filter for hyperbolic tangent waveshaping.

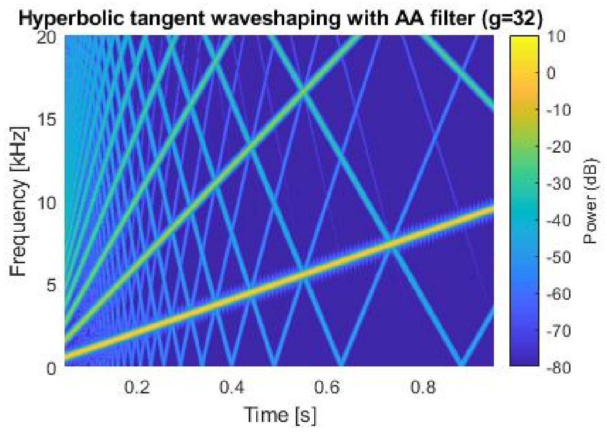


(a) Waveshaped chirp for $g = \frac{1}{2}\pi$, with anti-aliasing filter.

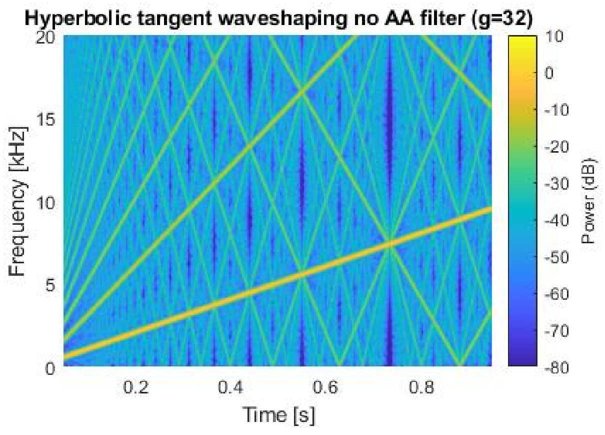


(b) Waveshaped chirp for $g = \frac{1}{2}\pi$, without the anti-aliasing filter.

Figure A.7: Comparing the effect of the anti-aliasing LP12 filter for sinusoidal waveshaping.

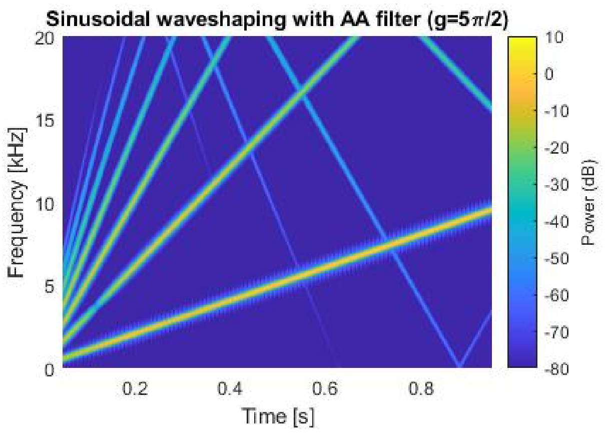


(a) Waveshaped chirp for $g = 32$, with anti-aliasing filter.

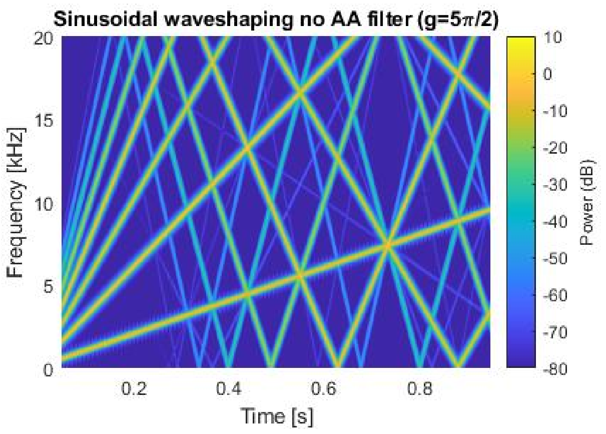


(b) Waveshaped chirp for $g = 32$, without the anti-aliasing filter.

Figure A.8: Comparing the effect of the anti-aliasing LP12 filter for hyperbolic tangent waveshaping.



(a) Waveshaped chirp for $g = \frac{5}{2}\pi$, with anti-aliasing filter.



(b) Waveshaped chirp for $g = \frac{5}{2}\pi$, without the anti-aliasing filter.

Figure A.9: Comparing the effect of the anti-aliasing LP12 filter for sinusoidal waveshaping.

Appendix B
Additional sections and information

B.1. MIDI

The MIDI protocol is traditionally a communications standard specified for electronic inter-instrument communication. The protocol has been specified for a variety of mediums, which include USB 1.0 and 2.0, data storage in ".mid" files (SMF), and the original UART-based protocol that interfaces with a 5-pin MIDI (DIN) connector (of which only 3 pins are used).

A MIDI message consists of 3 bytes [35]. The first byte contains status information, which indicates the message type. Messages can be sent on one of 16 channels, which are identified in the status byte if applicable. There are 5 message types:

1. Channel Voice: contains note information, such as note on/off, aftertouch, pressure and control change. This is the most important message for the application at hand.
2. Channel Mode: sound off; reset; all notes off.
3. System Common: manufacturer information; song select and position; tune request.
4. System Real-Time: intended for timing and sequencing, which includes the timing clock (24 times per quarter note) and start and stop sequencing.
5. System Exclusive (Sysex): specific to the device.

The Midifile library is used to convert SMFs into interpretable data that can be used to drive our system. The library is used to convert an SMF to note information that includes a starting and ending timestamp. To emulate the process that would occur on an MCU, the data is processed in fixed chunks of size 128, which is stored in a buffer and then written to a wav file. The code is shown in listing B.1 below.

All code is publicly available at https://github.com/marcorad/skripsie.

```
struct note {
  uint8_t note;
  float start_seconds;
  float duration_seconds;
  float vel;
};

std::vector<note> note_queue;


void load_notes(const std::string& file, uint8_t track, bool debug = false) {
  note_queue.clear();
  smf::MidiFile midi;
  midi.read(file);
  midi.linkEventPairs();
  midi.linkNotePairs();
  midi.doTimeAnalysis();
  float T = 1.0f / FS * PLAYBACK_BUFFER_SIZE;
  for (int i = 0; i < midi[track].size(); i++) {
    if (!midi[track][i].isNoteOn()) {
      continue;
    }
    auto note_on = midi[track][i];
    note n = { note_on[1], note_on.seconds, midi[track][i].getDurationInSeconds(),
   note_on[2] };
    note_queue.push_back(n);
    if(debug) std::cout << "note: " << (int)n.note << " t: " << n.start_seconds << "
   dt: " << n.duration_seconds << std::endl;
  }
  //sort according to trigger on
  std::sort(note_queue.begin(), note_queue.end(), [](const note& n1, const note& n2){
    return n1.start_seconds < n2.start_seconds;
    });
}

//simulates the PLAYBACK_BUF_SIZE nature of dealing with a queue of note events
void write_midi_to_wav(gen_manager* gm, gen_config* gc, const std::string& name, bool
     debug = false) {
  using namespace std;

  float dt = (float)PLAYBACK_BUFFER_SIZE / FS;
  float t = note_queue[0].start_seconds - 0.1f; //start time with 100ms space before
   starting
  float start_t = t;
  float end_t = note_queue.back().start_seconds + note_queue.back().duration_seconds
   + 1.0f; //end time with 1s space (for possible decays)
  uint32_t N = (uint32_t)( (end_t - t) * FS );

  float T = 1.0f / FS * PLAYBACK_BUFFER_SIZE; // time taken to play one buffer of
   samples

  float* bufL = new float[N];
  float* bufR = new float[N];
  uint32_t sample_index = 0;
  std::vector<note> playing = {};
  unsigned int i = 0;
  std::vector<note> trigger_off = {};
  while (t <= end_t) {
    //lambda to see if note must be triggered off
    auto l = [&](const note& n) {
      return n.start_seconds + n.duration_seconds <= t ;
    };
    //copy notes that must be triggered off
    std::copy_if(playing.begin(), playing.end(), std::back_inserter(trigger_off), l);
    //remove them
    playing.erase(std::remove_if(playing.begin(), playing.end(), l), playing.end());
    std::for_each(trigger_off.begin(), trigger_off.end(), [&](const note& n) {
      //TRIGGER OFF
      if (debug) cout << (n.start_seconds - start_t) << " note off: " << note_names[n
   .note] << endl;
      gm_trigger_note_off(gm, n.note);
      });
    trigger_off.clear();
    //find trigger on events
    while (i < note_queue.size()) {
      note n = note_queue[i];
      if (n.start_seconds < t) {
        playing.push_back(n);
        //TRIGGER ON
        if (debug) cout << (n.start_seconds - start_t) << " note on: " << note_names[n.note] << endl;
        gm_trigger_note_on(gm, gc, n.note, n.vel);
        i++;
      }
      else {
        break;
      }
    }

    //write samples
    uint32_t sample_size = std::min<uint32_t>(PLAYBACK_BUFFER_SIZE, N - sample_index);
    gm_write_n_samples(gm, gc, bufL + sample_index, bufR + sample_index, sample_size);
    //update time and index
    t += dt;
    sample_index += sample_size;
  }
  //write to file
  write_to_wav(name, bufL, bufR, N, FS);

}
```

Listing B.1: Simulating real-time MIDI input via SMF data.

Code was written to a wav file, using the correct wav file header information. 16-bit audio at 44.1 kHz is used. This is shown in listing B.2 below.

```
// WAVE PCM soundfile format (you can find more in http://soundfile.sapp.org/doc/WaveFormat/ )
typedef struct header_file
{
  char chunk_id[4] = { 'R','I','F','F'};
  int chunk_size; //TO BE FILLED IN:  36 + SubChunk2Size
  char format[4] = { 'W','A','V','E' };
  char subchunk1_id[4] = { 'f','m','t',' ' };
  int subchunk1_size = 16; //16 for PCM
  short int audio_format = 1; // PCM = 1
  short int num_channels = 2; //stereo
  int sample_rate;      //TO BE FILLED IN: sample_rate denotes the sampling rate.
  int byte_rate; //TO BE FILLED IN: SampleRate * NumChannels * BitsPerSample/8
  short int block_align = 2 * 16 / 8; //NumChannels * BitsPerSample/8
  short int bits_per_sample = 16; //bits per sample
  char subchunk2_id[4] = { 'd','a','t','a' };
  int subchunk2_size;     //TO BE FILLED IN: NumSamples * NumChannels * BitsPerSample/8
} header;

typedef struct header_file* header_p;
```

```

void write_to_wav(const std::string& name, float bufL[], float bufR[], int
    num_samples, int fs) {
  using namespace std;
  header head;
  head.sample_rate = fs;
  head.byte_rate = fs * 2 * 16 / 8;
  head.subchunk2_size = num_samples * 2 * 16 / 8;
  head.chunk_size = 36 + head.subchunk2_size;

  ofstream f;
  f.open("..\\wav\\" + name + ".wav", ios::binary);
  f.write(reinterpret_cast<char*> (&head), sizeof(header));

  for (int i = 0; i < num_samples; i++) {
    short int s = (int)(bufL[i] * (float)INT16_MAX);
    f.write(reinterpret_cast<char*> (&s), sizeof(short int));
    s = (int)(bufR[i] * (float)INT16_MAX);
    f.write(reinterpret_cast<char*> (&s), sizeof(short int));
  }
  f.close();
}
```

Listing B.2: Writing to a wav file.

B.2. A basic monophonic modular setup

Figure B.1 shows a block diagram of a typical modular setup. Many keyboard synthesisers follow this type topology.

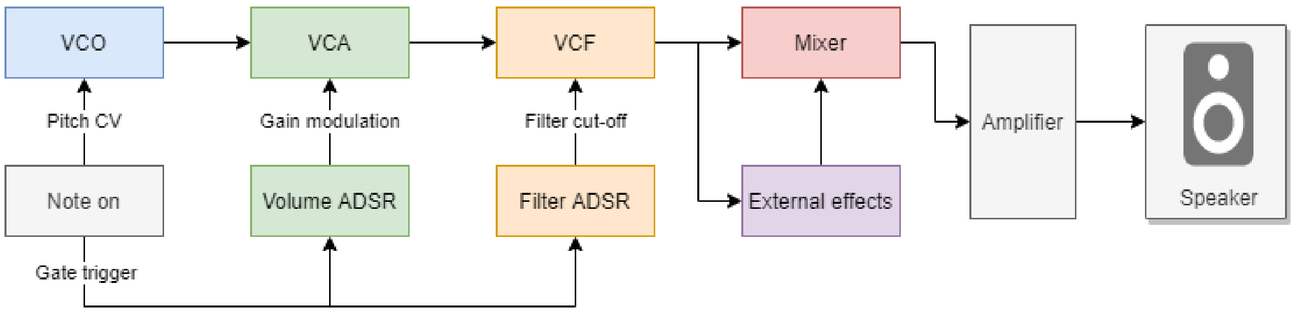


Figure B.1: Block diagram of a very basic monophonic modular setup.

The system consists of a note controller, which sends modulating signals to all of the modules. The modules have user-defined parameters, that is also modulated by ADSR sources.

The VCO generates the audio signal, which is then attenuated accordingly by the VCA and the volume ADSR envelope. The gain-modulated signal then enters the filter, which has a cut-off parameter that is modulated by a filter ADSR envelope. Note that the VCF and VCA can be swapped in order. The ADSR has very low-frequency content, which makes the order of these modules mostly inconsequential. The transient response of the VCF will be affected by the order.

The final modified signal is sent to a mixer, which can apply additional effects such as distortion, reverb or delay (either in series or parallel). These effects could be internally included in the synthesiser, or part of an FX loop. The amplification and speaker is not necessarily included, depending on the output of the synthesiser (could be a line-out, a digital signal within a DAW or internal speakers). It should be noted that many in-between steps is often included at the discretion of the designer. An example would be distortion/saturation between the VCO and VCA.

B.3. Regarding integer arithmetic using Q-numbers

The system in this thesis was partially written for integer arithmetic using Q-numbers, but then later rewritten due to a variety of factors.

Q-numbers is a way of representing fractional data using only integers, with an explicit denominator that is a power of 2. For example, a Q15 number is stored as $n_{Q15} = \frac{n}{2^{15}}$. This allows for storage of rational numbers, with a linear quantisation interval, whereas single-precision floating-point numbers have 23 bits of resolution for a power-of-2 associated exponent. Thus, floating point numbers have a varying quantisation over the numbers that they store, and can prove useful for dealing with small and large numbers alike.

Q-number multiplication is shown in the C-implementation below, which consists of casting, integer multiplication and bit-shift operations. The C-implementations for Q-number multiplication and division can be found at [36].

It should be noted that a 32-bit Q-number multiplication requires casts to a 64 bit integers and their multiplication. Thus, for allowing 24-bit audio resolution, arithmetic must be performed with 64-bit numbers, which is impractical on 32-bit processors and ends up being slower than floating-point FPU operations.

Reduced precision at lower values (such as values close to 0), also proved to be inaccurate when interpolating the sinusoidal LUT for filter coefficient calculation. The resulting filter coefficients at low cut-off frequencies were wildly inaccurate and caused unpredictable filter behaviour. Furthermore, the range is also limited by the Q-number. For example, a 16-bit signed Q15 number can only store values between -1 and +1 before overflowing. This can be problematic for very resonant filters, and requires the constant re-scaling of numbers so that it remains within range, which results in further precision loss and a higher noise floor. For signed numbers, it must also be guaranteed that bit-shifting compiles to an arithmetic shift ASM operation, which can be compiler dependent, making it unreliable for porting to different platforms [37].

An FPU floating point operation is less expensive than a Q-number multiplication, since an FPU operation requires 1 clock cycle (see table C.3) to load data from processor to FPU and 1 clock cycle to retrieve. Thus, a multiplication requires, at most, 3 clock cycles. This is reduced if data does not need to be shifted to and from the main processor.

Q-number multiplication performance can also be comparable to FPU multiplication, if saturation is not included. However, the lack of 24-bit audio resolution for 32-bit systems without requiring 64-bit integers and the limited precision of Q-number for smaller values and its range restrictions, resulted in floating point being chosen for the system, requiring a refactor of the initial implementation that used Q-numbers.

B.4. Optimising harmonic indexing using a LUT

We are interested in creating a LUT for equation B.4, shown below for convenience.

$$i = \min(\lfloor \log_2(\lfloor h_{max} \rfloor) \rfloor, K)$$

We can reduce the memory consumption of the harmonic indexing LUT by 4, by considering the equation:

$$i = \lfloor \log_2(\lfloor \frac{h_{max}}{4} \rfloor) \rfloor$$

This LUT can easily be constructed. Since we only 2^{b-1} harmonics within the buffer, we originally need a LUT of size 2^{b-1}. This technique allows for a LUT of size 2^{b-3}. For this thesis, we use $b = 9$, hence we originally need a LUT of size 256. For this technique, we only need a LUT of size 64, and extra indexing code. This technique was verified to yield correct results using Microsoft Excel.

The harmonic indexing equation then becomes:

$$i = \begin{cases} 0, \; h_{max} \leq 2 \\ 1, \; 2 < h_{max} \leq 4 \\ \lfloor \log_2(\lfloor \frac{h_{max}}{4} \rfloor) \rfloor, \; h_{max} > 4 \end{cases} \tag{B.1}$$

Note that larger memory reduction is possible, but would require more logic to handle the additional piece-wise functions added to i.

B.5. Alternative uses and signal chains for the designed components

The choice of processing order in the generator component for the design in this thesis is arbitrary, and depends on the designer. Order matters greatly, since this system is inherently non-linear, caused by waveshaping and ADSR volume modulation. Figure B.2 shows the signal chain as designed in this thesis, for reference. It should be noted that 3 oscillators is the minimum recommended for stereo width, but any odd number of oscillators greater than this can further add a greater stereo effect.

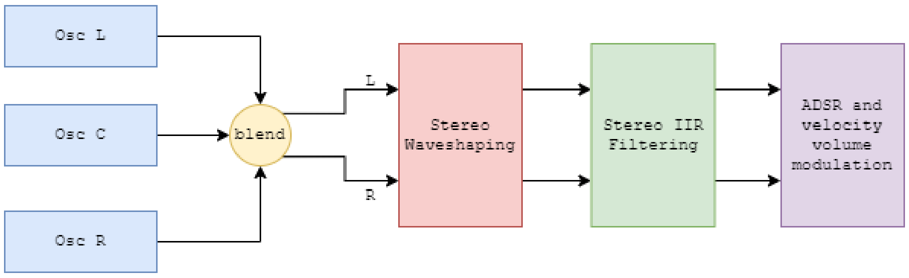


Figure B.2: The current signal chain, as designed in this thesis.

The placement of the waveshaping component can greatly influence the effect it has on the audio. From figure B.2, the left and right channels are waveshaped. This implies that the oscillators are coupled in the waveshaping process, i.e, a change in behaviour oscillators, such as by detuning our stereo width modification, will greatly impact the results. As such, figure B.3 shows alternative placements for the waveshaping component. If the waveshaping is placed last, it will act as a form of distortion/saturation. This makes the results highly dependent on filter, volume and stereo settings. This effect is often desirable, but could require the introduction of more aliasing, as the performer could require a highly distorted signal.

Alternatively, the waveshaping can be placed after each oscillator, which would decouple any effects caused by stereo blending. This would lead to more predictable results, but comes at the cost of more waveshaping components, increasing computational requirements. This would not scale well in the addition of more oscillators in this system.

Furthermore, hyperbolic tangent waveshaping can be used in the feedback loop of the filter, shown in figure B.4. This mimics the behaviour of analogue tube filters, which often becomes distorted for higher Q values. The non-linearity of tube filters can be reflected here, since it will add additional harmonics to the signal, which is often desired by performers that extensively utilises filters.

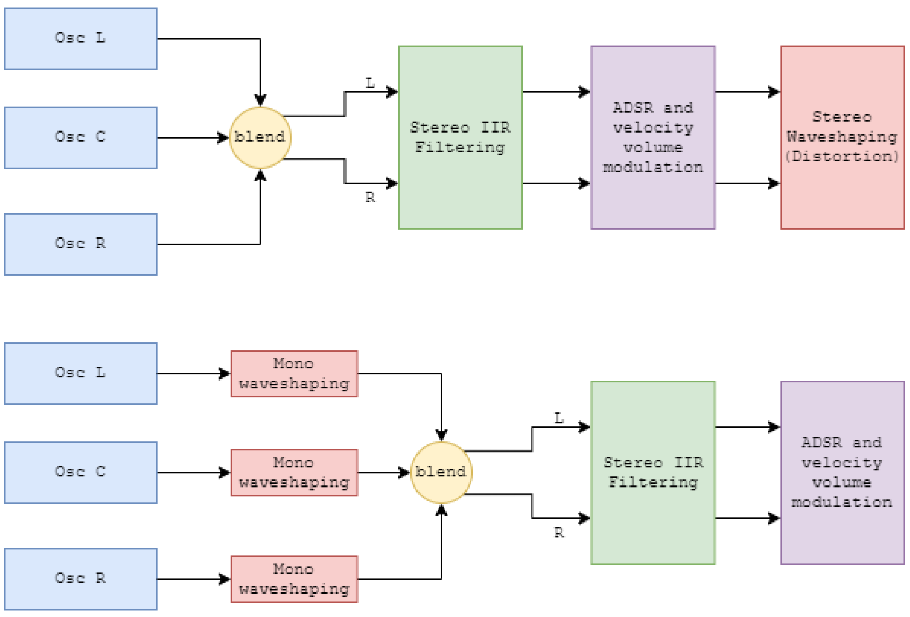


Figure B.3: Alternative signal chain placement for waveshaping.

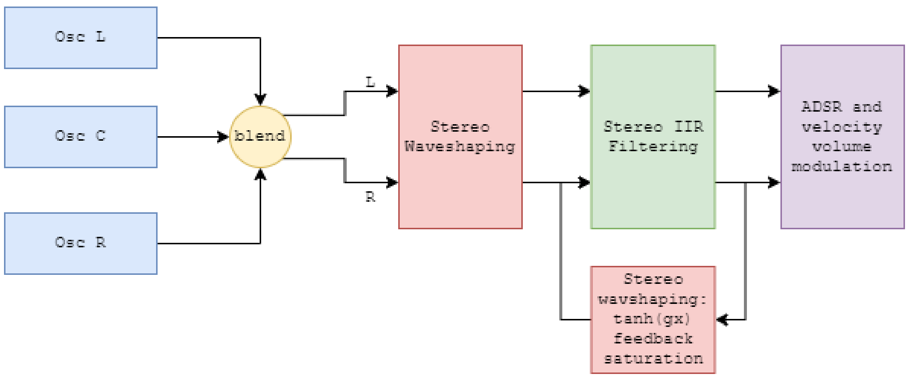

Osc L
Osc C
Osc R
blend
L
R
Stereo Waveshaping
Stereo IIR Filtering
ADSR and velocity volume modulation
Stereo wavshaping: tanh(gx) feedback saturation

Figure B.4: Additional feedback saturation in the IIR filters.

Figures B.2, B.3 and B.4 can be combined in any way to also achieve another alternative signal chain. More filters, waveshapers and oscillators can be introduced to create a more complicated product, using the building blocks that were designed in this thesis.

B.6. Musical test

TESTING STUFF

Appendix C
Tables

Table C.1: Bilinear transform substitution [2]

S-domain	Z-domain
1	$(1+2z^{-1}+z^{-2})(1-\cos(\omega_0))$
s	$(1-z^{-2})\sin(\omega_0)$
s^2	$(1-2z^{-1}+z^{-2})(1+\cos(\omega_0))$
$1+s^2$	$2(1-2\cos(\omega_0)z^{-1}+z^{-2})$

Table C.2: MIDI note IDs and their frequencies [3]

MIDI (hexadecimal)	Note Name	Frequency (Hz)
0x0		8.18
0x1		8.66
0x2		9.18
0x3		9.72
0x4		10.3
0x5		10.91
0x6		11.56
0x7		12.25
0x8		12.98
0x9		13.75
0xA		14.57
0xB		15.43
0xC		16.35
0xD		17.32
0xE		18.35
0xF		19.45
0x10		20.6
0x11		21.83
0x12		23.12
0x13		24.5
0x14		25.96
0x15	A0	27.5
0x16	A#0/Bb0	29.14
0x17	B0	30.87
0x18	C1	32.7
0x19	C#1/Db1	34.65
0x1A	D1	36.71
0x1B	D#1/Eb1	38.89
0x1C	E1	41.2

0x1D	F1	43.65
0x1E	F#1/Gb1	46.25
0x1F	G1	49
0x20	G#1/Ab1	51.91
0x21	A1	55
0x22	A#1/Bb1	58.27
0x23	B1	61.74
0x24	C2	65.41
0x25	C#2/Db2	69.3
0x26	D2	73.42
0x27	D#2/Eb2	77.78
0x28	E2	82.41
0x29	F2	87.31
0x2A	F#2/Gb2	92.5
0x2B	G2	98
0x2C	G#2/Ab2	103.83
0x2D	A2	110
0x2E	A#2/Bb2	116.54
0x2F	B2	123.47
0x30	C3	130.81
0x31	C#3/Db3	138.59
0x32	D3	146.83
0x33	D#3/Eb3	155.56
0x34	E3	164.81
0x35	F3	174.61
0x36	F#3/Gb3	185
0x37	G3	196
0x38	G#3/Ab3	207.65
0x39	A3	220
0x3A	A#3/Bb3	233.08
0x3B	B3	246.94
0x3C	C4 (middle C)	261.63
0x3D	C#4/Db4	277.18
0x3E	D4	293.66
0x3F	D#4/Eb4	311.13
0x40	E4	329.63
0x41	F4	349.23
0x42	F#4/Gb4	369.99
0x43	G4	392
0x44	G#4/Ab4	415.3
0x45	A4 concert pitch	440
0x46	A#4/Bb4	466.16
0x47	B4	493.88
0x48	C5	523.25

0x49	C#5/Db5	554.37
0x4A	D5	587.33
0x4B	D#5/Eb5	622.25
0x4C	E5	659.26
0x4D	F5	698.46
0x4E	F#5/Gb5	739.99
0x4F	G5	783.99
0x50	G#5/Ab5	830.61
0x51	A5	880
0x52	A#5/Bb5	932.33
0x53	B5	987.77
0x54	C6	1046.5
0x55	C#6/Db6	1108.73
0x56	D6	1174.66
0x57	D#6/Eb6	1244.51
0x58	E6	1318.51
0x59	F6	1396.91
0x5A	F#6/Gb6	1479.98
0x5B	G6	1567.98
0x5C	G#6/Ab6	1661.22
0x5D	A6	1760
0x5E	A#6/Bb6	1864.66
0x5F	B6	1975.53
0x60	C7	2093
0x61	C#7/Db7	2217.46
0x62	D7	2349.32
0x63	D#7/Eb7	2489.02
0x64	E7	2637.02
0x65	F7	2793.83
0x66	F#7/Gb7	2959.96
0x67	G7	3135.96
0x68	G#7/Ab7	3322.44
0x69	A7	3520
0x6A	A#7/Bb7	3729.31
0x6B	B7	3951.07
0x6C	C8	4186.01
0x6D	C#8/Db8	4434.92
0x6E	D8	4698.64
0x6F	D#8/Eb8	4978.03
0x70	E8	5274.04
0x71	F8	5587.65
0x72	F#8/Gb8	5919.91
0x73	G8	6271.93
0x74	G#8/Ab8	6644.88

0x75	A8	7040
0x76	A#8/Bb8	7458.62
0x77	B8	7902.13
0x78	C9	8372.02
0x79	C#9/Db9	8869.84
0x7A	D9	9397.27
0x7B	D#9/Eb9	9956.06
0x7C	E9	10548.08
0x7D	F9	11175.3
0x7E	F#9/Gb9	11839.82
0x7F	G9	12543.85

Table C.3: ARM Cortex M4 and M7 FPU instruction set [4]

Operation	Assembler	Cycles
Absolute value	VABS.F32	1
Addition	VADD.F32	1
Compare	VCMP.F32	1
	VCMPE.F32	1
Convert	VCVT.F32	1
Divide	VDIV.F32	14
Load	VLDM.64	1+2*N
	VLDM.32	1+N
	VLDR.64	3
	VLDR.32	2
Move	VMOV	1
	VMOV	1
	VMOV	2
	VMRS	1
	VMSR	1
Multiply	VMUL.F32	1
	VMLA.F32	3
	VMLS.F32	3
	VNMLA.F32	3
	VNMLS.F32	3
Multiply (fused)	VFMA.F32	3
	VFMS.F32	3
	VFNMA.F32	3
	VFNMS.F32	3
Negate	VNEG.F32	1
	VNMUL.F32	1
Pop	VPOP.64	1+2*N
	VPOP.32	1+N
Push	VPUSH.64	1+2*N
	VPUSH.32	1+N
Square-root	VSQRT.F32	14
Store	VSTM.64	1+2*N
	VSTM.32	1+N
	VSTR.64	3
	VSTR.32	2
Subtract	VSUB.F32	1

Appendix D

Code listings

All code is publicly available at https://github.com/marcorad/skripsie.

```c
float cents_scaling_factor[] = {
1.00000000000000f,1.00057778950655f,1.00115591285382f,1.00173437023470,
1.00231316184217f,1.00289228786937f,1.00347174850950f,1.00405154395592,
1.00463167440205f,1.00521214004148f,1.00579294106785f,1.00637407767497,
1.00695555005672f,1.00753735840711f,1.00811950292026f,1.00870198379040,
1.00928480121187f,1.00986795537914f,1.01045144648676f,1.01103527472943,
1.01161944030192f,1.01220394339916f,1.01278878421615f,1.01337396294802,
1.01395947979003f,1.01454533493752f,1.01513152858597f,1.01571806093096,
1.01630493216819f,1.01689214249346f,1.01747969210269f,1.01806758119192,
1.01865580995729f,1.01924437859508f,1.01983328730164f,1.02042253627348,
1.02101212570719f,1.02160205579949f,1.02219232674721f,1.02278293874728,
1.02337389199677f,1.02396518669285f,1.02455682303280f,1.02514880121402,
1.02574112143402f,1.02633378389042f,1.02692678878098f,1.02752013630354,
1.02811382665607f,1.02870786003665f,1.02930223664349f,1.02989695667490,
1.03049202032930f,1.03108742780523f,1.03168317930136f,1.03227927501645,
1.03287571514939f,1.03347249989918f,1.03406962946493f,1.03466710404588,
1.03526492384138f,1.03586308905088f,1.03646159987396f,1.03706045651031,
1.03765965915975f,1.03825920802219f,1.03885910329766f,1.03945934518634,
1.04005993388848f,1.04066086960447f,1.04126215253481f,1.04186378288011,
1.04246576084112f,1.04306808661868f,1.04367076041375f,1.04427378242741,
1.04487715286087f,1.04548087191543f,1.04608493979253f,1.04668935669371,
1.04729412282063f,1.04789923837507f,1.04850470355893f,1.04911051857422,
1.04971668362307f,1.05032319890772f,1.05093006463054f,1.05153728099401,
1.05214484820072f,1.05275276645338f,1.05336103595484f,1.05396965690802,
1.05457862951601f,1.05518795398198f,1.05579763050924f,1.05640765930119,
1.05701804056138f,1.05762877449346f,1.05823986130119f,1.05885130118847,
1.05946309435930
};

float semitones_scaling_factor[] = {
1.00000000000000f,1.05946309435930f,1.12246204830937f,1.18920711500272,
1.25992104989487f,1.33483985417003f,1.41421356237310f,1.49830707687668,
1.58740105196820f,1.68179283050743f,1.78179743628068f,1.88774862536339
};

float get_detune_factor_semitones_lut(int st) {
  int i = abs(st);
  int oct = i / 12;
  i = i - oct * 12;
  return ((float)(1<<oct)) * semitones_scaling_factor[i];
}

float get_detune_factor_cents_lut(float cents) {
  float xc = fabs(cents);
  float xs = floorf(xc * 0.01f); //amount of integer semitones
  xc = ( xc - 100.0f * xs); //cents in the range [0,100)
  float stf = get_detune_factor_semitones_lut((int)xs); //scaling from octaves
  float cf = lut_lookup_no_wrap(cents_scaling_factor, xc); //scaling from cents,
    interpolated
  float f = stf * cf; //cumulative scaling factor
  return cents < 0.0f ? 1.0f / f : f; //scaling up or down
}

float get_detune_factor(float cents) {
  return get_detune_factor_cents_lut(cents);
}

float detune_cents(float orig_freq, float cents) {
  return orig_freq * get_detune_factor(cents);
}
```

Listing D.1: Efficient frequency scaling using the equal temperament tuning system

```c
inline void wt_config_digital_freq(wavetable* wt, float freq) {
  wt->stride = freq * (float)LUT_SIZE; //set strides, with no FM applied
  wt->base_stride = wt->stride;
  uint16_t harmonics = (uint16_t)(0.5f / freq);
  harmonics = harmonics > (LUT_SIZE>>1) ? (LUT_SIZE>>1) : harmonics;
  uint8_t harmonic_index = harmonics < 4 ? 0 : harmonic_indices[(harmonics>>2)-1];
  wt->harmonic_index = harmonic_index;
}
```

Listing D.2: Configuring wavetable frequency

```c
void iir_calc_lp24_coeff(IIR_coeff* filter, float f0, float q) {
  float cosw = cos_lookup(f0);
  float sinw = sin_lookup(f0);
  float twoq = 2.0f * q;
  //normalising by a0_recip = 1 / a0
  float a0_recip = 1.0f / (twoq + sinw); //division is unavoidable here
  float a1 = 2.0 * twoq * cosw; //pre-negated: d1 = -a1/a0
  float a2 = sinw - twoq; //pre-negated: d2 = -a2/a0
  float b0, b1, b2;
  b1 = twoq * (1.0f - cosw);
  b0 = b1 * 0.5f;
  b2 = b0;
  filter->n0 = b0 * a0_recip;
  filter->n1 = b1 * a0_recip;
  filter->n2 = b2 * a0_recip;
  filter->d1 = a1 * a0_recip;
  filter->d2 = a2 * a0_recip;
}
```

Listing D.3: Calculating the coefficients for a LP24 filter

```c
inline float iir_filter_sample(IIR_coeff* coeff, IIR_prev_values* prev, float x) {
  float y = coeff->n0 * x;
  y += coeff->n1 * prev->xm1; //FPU VFMA.F32 operations
  y += coeff->n2 * prev->xm2;
  y += coeff->d1 * prev->ym1;
  y += coeff->d2 * prev->ym2;
  prev->ym2 = prev->ym1; //unit delays
  prev->ym1 = y;
  prev->xm2 = prev->xm1;
  prev->xm1 = x;
  return prev->ym1;
}
```

Listing D.4: Filtering a signal

```c
//approximates cos(2*pi*f) for f in [0,0.75)
inline float cos_lookup(float f) {
    return lut_lookup(basic_luts[SINE][0] + LUT_SIZE / 4, LUT_SIZE, f * (float)
    LUT_SIZE);
}
//approximates sin(2*pi*f) for f in [0,1)
inline float sin_lookup(float f) {
    return lut_lookup(basic_luts[SINE][0], LUT_SIZE, f * (float)LUT_SIZE);
}
```

Listing D.5: Trigonometric lookup functions

```c
inline float adsr_sample(ADSR* adsr, float params[]) {
  if (adsr->state == SUSTAIN) {
    return params[SUSTAIN];
  }
  if ((adsr->phase < (float)EXP_LUT_SIZE - 1.0f))
  { //no transition
    float lookup = lut_lookup(lut_exp, EXP_LUT_SIZE, adsr->phase);
    adsr->prev_sample = adsr->scale * lookup + adsr->offset;
  }
  else { //transition
    adsr->state += 1;
    adsr->phase = 0.0f;
    //calculate scaling and offset
    if (adsr->state == DECAY) {
      adsr->scale = params[SUSTAIN] - adsr->prev_sample;
      adsr->offset = adsr->prev_sample;
    }
    else if (adsr->state == SUSTAIN) {
      adsr->prev_sample = params[SUSTAIN];
      return params[SUSTAIN];
    }
  }
  if (adsr->state != NOT_PLAYING) {
  //increment phase
    adsr->phase += params[adsr->state];
    return adsr->prev_sample;
  }
  else {
  //return 0 if not playing
    adsr->prev_sample = 0.0f;
  }
  return adsr->prev_sample;
}
```

Listing D.6: Sampling from an ADSR envelope state-machine

```
inline void gen_vibrato(generator* g, gen_config* gc) {
  float osc = (wt_sample_no_interpolation(&g->wt_vibrato, 0));
  float ampl = (gc->vibrato_factor - 1.0f) * g->base_freq;
  float vib = osc * ampl;
  wt_apply_fm(&g->wt_left, vib);
  wt_apply_fm(&g->wt_right, vib);
  wt_apply_fm(&g->wt_center, vib);
}
```

Listing D.7: Applying vibrato FM

```
inline void gen_sample(generator* g, gen_config* gc, float* buf_L, float* buf_R) {
  //apply vibrato using FM
  gen_vibrato(g, gc);
  //sample L, R, C channels
  float sc = wt_sample(&g->wt_center, gc);
  float sl = wt_sample(&g->wt_left, gc);
  float sr = wt_sample(&g->wt_right, gc);
  //blend detuned samples
  float width = 1.0f - gc->detune_width; //invert to get volume in other channel
  //detune_width of 1 seperates sr and sl into L and R channels
  //detune_width of 0 is mono
  float L = sc + gc->detune_volume * (sl + width * sr);
  float R = sc + gc->detune_volume * (sr + width * sl);
  //find f0 using envelope
  float f0 = g->filter_freq_start + adsr_sample(&g->envelope_filter_cutoff, gc->
    filt_adsr_params) * g->filter_envelope_amplitude;
  f0 = clamp(f0, DIGITAL_FREQ_20HZ, DIGITAL_FREQ_20KHZ); //limit to audible range
  //calculate filter coefficients
  (*(gc->filter_coeff_func))(&g->filter_coeff, f0, gc->filter_Q);
  //waveshape L and R
  L = gen_waveshape_sample(&g->filter_sat_pv_L, gc, L);
  R = gen_waveshape_sample(&g->filter_sat_pv_R, gc, R);
  //apply volume
  float volume = adsr_sample(&g->envelope_volume, gc->vol_adsr_params) * g->velocity;
  L = L * volume;
  R = R * volume;
  //filter channels
  L = iir_filter_sample(&g->filter_coeff, &g->filter_left_pv, L);
  R = iir_filter_sample(&g->filter_coeff, &g->filter_right_pv, R);
  //output
  *buf_L = L;
  *buf_R = R;
}
```

Listing D.8: Sampling from a generator

```
generator* note_played_hash_table[128];

struct gen_manager {
  generator generators[NUM_GENERATORS];
  generator* available[NUM_GENERATORS];
  int8_t available_head = NUM_GENERATORS;
  generator* in_use[NUM_GENERATORS];
  uint8_t in_use_head = 0;
};
```

```

inline void gm_init(gen_manager* gm) {
  //make all available
  for (int i = 0; i < NUM_GENERATORS; i++)
  {
    gm->available[i] = &gm->generators[i];
  }
  gm->available_head = NUM_GENERATORS;
  gm->in_use_head = 0;
  //none are in use
  for (int i = 0; i < NUM_GENERATORS; i++)
  {
    gm->in_use[i] = nullptr;
  }
  //none are playing notes
  for (int i = 0; i < 128; i++) {
    note_played_hash_table[i] = nullptr;
  }
}

inline void gm_add_to_in_use(gen_manager* gm, generator* g) {
  gm->in_use[gm->in_use_head] = g;
  gm->in_use_head++;
}

inline generator* gm_pop_off_back_from_in_use(gen_manager* gm) {
  generator* g = gm->in_use[0];
  //shift queue
  for (int i = 0; i < gm->in_use_head; i++)
  {
    gm->in_use[i] = gm->in_use[i + 1];
  }
  gm->in_use_head--;
  return g;
}

inline void gm_add_to_available(gen_manager* gm, generator* g) {
  gm->available[gm->available_head] = g;
  gm->available_head++;
}

inline generator* gm_pop_off_front_from_available(gen_manager* gm) {
  gm->available_head--;
  return gm->available[gm->available_head];
}

inline void gm_set_gen_playing_note(generator* g, uint8_t midi_note) {
  note_played_hash_table[midi_note] = g;
}

//get an available gen, or the oldest playing gen if none are available
inline generator* gm_get_gen(gen_manager* gm) {
  generator* g;
  if (!gm->available_head == 0) {
    //pop off front of available
    g = gm_pop_off_front_from_available(gm);
  }
  else {//none are available (this should be a rare case of key-mashing) or long release times
    g = gm_pop_off_back_from_in_use(gm);
    gm_set_gen_playing_note(nullptr, g->midi_note); //make sure that the retrigger of a note-off will not affect the new note
  }
  gm_add_to_in_use(gm, g);
  return g;
}

//add all non-playing gens to the available list in O(N) time.
inline void gm_make_not_playing_available(gen_manager* gm) {
  int count = 0;
  for (int i = 0; i < gm->in_use_head; i++)
  {
    generator* g = gm->in_use[i];
    if (!gen_is_playing(g)) {
      count++;
      gm_add_to_available(gm, g);
      gm_set_gen_playing_note(nullptr, g->midi_note); //remove from hashtable
    }
    else {
      gm->in_use[i - count] = g;
    }
  }
  gm->in_use_head -= count;
}

inline generator* gm_get_gen_playing_note(uint8_t midi_note) {
  return note_played_hash_table[midi_note];
}

float L_temp[PLAYBACK_BUFFER_SIZE];
float R_temp[PLAYBACK_BUFFER_SIZE];

inline void gm_write_n_samples(gen_manager* gm, gen_config* gc, float bufL[], float bufR[], uint32_t n) {
  for (int i = 0; i < PLAYBACK_BUFFER_SIZE; i++) //init to zero for addition later
  {
    bufL[i] = 0.0f;
    bufR[i] = 0.0f;
  }
  for (int i = 0; i < gm->in_use_head; i++) //for all active voices
  {
    gen_write_n_samples(gm->in_use[i], gc, L_temp, R_temp, n); //write to temp
    for (int i = 0; i < PLAYBACK_BUFFER_SIZE; i++) //for all samples in each voice
    {
      bufL[i] += 0.125f * L_temp[i]; //1/8th for 8 voices to ensure headroom
      bufR[i] += 0.125f * R_temp[i];
    }
  }
  for (int i = 0; i < PLAYBACK_BUFFER_SIZE; i++) //clamp to +- 1
  {
    float L = bufL[i], R = bufR[i];
    bufL[i] = clamp(L, -1.0f, 1.0f);
    bufR[i] = clamp(R, -1.0f, 1.0f);
  }
  //make generators that finished decay phase available
  gm_make_not_playing_available(gm);
}

inline void gm_trigger_note_on(gen_manager* gm, gen_config* gc, uint8_t note, uint8_t
     vel){
  generator* g = gm_get_gen_playing_note(note);
  if (g == nullptr) g = gm_get_gen(gm); //prevents any weirdness in retriggers of
    notes before a note off
  gen_freq(g, gc, notes_digital_freq[note], vel);
  g->midi_note = note;
  gm_set_gen_playing_note(g, note);
  gen_note_on(g);
}

inline void gm_trigger_note_off(gen_manager* gm, uint8_t note) {
  generator* g = gm_get_gen_playing_note(note);
  gm_set_gen_playing_note(nullptr, note);
  if (g != nullptr) gen_note_off(g); //prevents any weirdness in note off triggers if
     it's not actually on
}
```

Listing D.9: Generator manager functions

Appendix E

Test code

All code is publicly available at https://github.com/marcorad/skripsie.

```
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <random>

#include "LUT.h"
#include "init_luts.h"
#include "ADSR.h"
#include "IIR.h"
#include "wavetable.h"
#include "generator.h"
#include "play_notes.h"
#include "wav.h"


using namespace std;

//white noise generator
void generate_wn(float* buf, int N) {
  unsigned seed = 42; //LOL
  std::default_random_engine generator(seed);
  auto dist = std::normal_distribution<float>(0.0f, 1.0f);

  for (int i = 0; i < N; i++)
  {
    buf[i] = dist(generator);
  }
}

//chirp generator (in digital frequency)
void generate_linear_chirp_frequencies(float* buf, int N, float start, float end) {
  for (int i = 0; i < N; i++)
  {
    float x = (float)i / (float)N;
    float f = start + (end-start) * x;
    buf[i] = f;
  }
}

void add_wn(float* wn, int N, float u, float s) {
  srand(time(NULL));
  unsigned seed = rand();
  std::default_random_engine generator(seed);
  auto dist = std::normal_distribution<float>(u, s);

  for (int i = 0; i < N; i++)
  {
    wn[i] += dist(generator);
  }
}

void test_adsr(float* buf, int N) {
  ADSR adsr;
  int off = 0;
  int k = N / 8;
  float t = k / FS;
  float params[4];
  adsr_config(params, t, t, 0.5f, t);

  for (int i = 0; i < k / 3; i++)
  {
    buf[i] = 0.0f;
  }
  off += k / 3;
  adsr_trigger_on(&adsr);

  for (int i = off; i < off + k + k/2; i++)
  {
    buf[i] = adsr_sample(&adsr, params);
  }
  adsr_trigger_on(&adsr);
  off += k + k / 2;
  for (int i = off; i < off + 3 * k; i++)
  {
    buf[i] = adsr_sample(&adsr, params);
  }
  off += 3* k;
  adsr_trigger_off(&adsr);
  for (int i = off; i < off + N - off; i++)
  {
    buf[i] = adsr_sample(&adsr, params);
  }
}

void test_wavetable(float* buf, int N, float pos) {
  //wavetables at different pos and freq
  float f0 = notes_digital_freq[A0];
  int off = 0;
  int n = N / 8;
  gen_config gc;
  gc.wt_pos = pos;

  wavetable wt;

  for (float f = f0; f <= notes_digital_freq[A8]; f *= 2.0f) //sweep octaves
  {
    wt.phase = 0;
    wt_config_digital_freq(&wt, f);
    for (int i = off;  i < off + n; i++)
    {
      buf[i] = wt_sample(&wt, &gc);
    }
    off += n;
  }

}

void test_vibrato(float* buf, int N) {
  float f0 = 10000.0f/FS;
  float cents = 500.0f;
  generator g;
  gen_config gc;
  gen_config_default(&gc);

  gen_config_vibrato(&gc, cents, 2.0f / FS);

  gen_apply_vibrato_config(&g, &gc);
  gen_freq(&g, &gc, f0, 127);
  gen_note_on(&g);



  for (int i = 0; i < N; i++)
  {
    float temp;
    gen_sample(&g, &gc, buf + i, &temp);
  }
}

void test_filter(float* wn, float* out, float* adsr_out, int N, float a, float d,
    float s, float r, float freqstart, float freqend, float q, void (*
    filter_coeff_func)(IIR_coeff*, float, float)) {
  ADSR adsr;
  IIR_prev_values pv = {0,0,0,0};
  IIR_coeff coeff;
  float params[4];
  adsr_config(params, a, d, s, r);
  adsr_trigger_on(&adsr);

  for (int i = 0; i < N/2; i++)
  {
    float f0 = freqstart + adsr_sample(&adsr, params) * (freqend - freqstart);
    (*filter_coeff_func)(&coeff, f0, q);
    out[i] = iir_filter_sample(&coeff, &pv, wn[i]);
    adsr_out[i] = f0;
  }

  adsr_trigger_off(&adsr);

  for (int i = N/2; i < N; i++)
  {
    float f0 = freqstart + adsr_sample(&adsr, params) * (freqend - freqstart);
    (*filter_coeff_func)(&coeff, f0, q);
    out[i] = iir_filter_sample(&coeff, &pv, wn[i]);
    adsr_out[i] = f0;
  }
}

void test_waveshape(float* buf_tanh, float* buf_sin, int N, float* freq, float g_tanh
    , float g_sin, bool use_aa = true) {
  generator g;
  gen_config gc;
  gen_config_default(&gc);
  gen_apply_vibrato_config(&g, &gc);
  gen_note_on(&g);
  gen_config_tanh_saturator(&gc, g_tanh);
  gen_config_filter_envelope(&gc, 0.1f, 0.1f, 1.0f, 0.1f, 512.0f, 512.0f, 0.7071f);
  gen_config_wavetables(&gc, 0.0f, 0.0f, 0.0f, 0.0f);
  if(!use_aa)
    iir_calc_lp12_coeff(&gc.filter_sat_AA, DIGITAL_FREQ_20KHZ, 1.0f);
  for (int i = 0; i < N; i++)
  {
    float temp;
    gen_freq(&g, &gc, freq[i], 127);
    gen_sample(&g, &gc, buf_tanh + i, &temp);
  }

  //TODO IMPLEMENT SIN
  gen_config_sine_saturator(&gc, g_sin);
  gen_reset_AA_pv(&g); //reset pv from previous step
  if (!use_aa)
    iir_calc_lp12_coeff(&gc.filter_sat_AA, DIGITAL_FREQ_20KHZ, 1.0f);
  for (int i = 0; i < N; i++)
  {
    //WHERE IS THE NOISE COMING FROM?!?!?!?!?!
    float temp;
    gen_freq(&g, &gc, freq[i], 127);
    gen_sample(&g, &gc, buf_sin + i, &temp);
  }
}

void test_wavetable_sweep(float* buf, int N, float* freq, float pos) {
  wavetable wt;
  gen_config gc;

  gc.wt_pos = pos;

  for (int i = 0; i < N; i++)
  {
    wt_config_digital_freq(&wt, freq[i]);
    buf[i] = wt_sample(&wt, &gc);
  }
}

void test_stereo_width(float* bufL, float* bufR, int N, float width) {
  float f0 = 100.0f / FS;
  float cents = 1200.0f;
  generator g;
  gen_config gc;
  gen_config_default(&gc);
  gen_apply_vibrato_config(&g, &gc);
  gen_config_wavetables(&gc, 0.0f, cents, 1.0f, width);
  gen_freq(&g, &gc, f0, 127);
  gen_note_on(&g);
  gen_write_n_samples(&g, &gc, bufL, bufR, N);
}

void test_gm_overload(float buf[], int N) {

  gen_manager gm;
  gen_config gc;
  gen_config_default(&gc);
  gm_init(&gm);
  gc.filter_coeff_func = &iir_calc_no_coeff;
  gen_config_wavetables(&gc, 0.0f, 0.0f, 0.0f, 0.0f);

  float* temp = new float[N];

  float freq[16];
  for (int i = 0; i < 8; i++)
  {
    freq[i] = (1000.0f + i * 2000.0f) / FS; //1k, 3k, ..., 15k
    freq[i+8] = (2000.0f + i * 2000.0f) / FS; //2k, 4k, ..., 16k
  }

  int L = N / 16;
  int end = L * 16;
  int note = 0;
  for (int off = 0; off < end; off += L)
  {
    gm_trigger_note_on_freq(&gm, &gc, note, 127, freq[note]);
    gm_write_n_samples(&gm, &gc, buf + off, temp + off, L);
    note++;
  }

  //fill remaining with 0
  for (int i = 0; i < N-end; i++)
  {
    buf[end + i] = 0.0f;
  }
}



void test_gm_operation(float buf[], int N) {

  gen_manager gm;
  gen_config gc;
  gen_config_default(&gc);
  gm_init(&gm);
  gc.filter_coeff_func = &iir_calc_no_coeff;
  gen_config_wavetables(&gc, 0.0f, 0.0f, 0.0f, 0.0f);

  float* temp = new float[N];


  float freq[8];
  for (int i = 0; i < 8; i++)
  {
    freq[i] = (500.0f + i * 2500.0f) / FS; //0k5, 3k, 5k5, ..., 18k
  }

  int L = N / 16;
  int end = L * 16;
  int note = 0;
  int count = 0;
  //trigger all on, then trigger all off by LIFO
  for (int off = 0; off < end; off += L)
  {
    if (count < 8)
    {
      gm_trigger_note_on_freq(&gm, &gc, note, 127, freq[note]);
      note++;
    }
    else {
      gm_trigger_note_off(&gm, note);
      note--;
    }

    gm_write_n_samples(&gm, &gc, buf + off, temp + off, L);
    count++;
  }

  //fill remaining with 0
  for (int i = 0; i < N - end; i++)
  {
    buf[end + i] = 0.0f;
  }
}

void test_freq_scaling() {
  cout << "Detune factor 125.57: " << get_detune_factor_cents_lut(125.57f) << endl;
}

void print_to_file(float arr[], int size, const string& name) {
  ofstream f;
  f.open(name);
  f.precision(10);
  for (int i = 0; i < size; i++) {
    f << arr[i] << endl;
  }
  f.close();
}


```

```
void print_iir_coeff(IIR_coeff& f) {
  cout << "[" << f.n0 << ", " << f.n1 << ", " << f.n2 << "], [ 1, -" << f.d1 << ", -"
     << f.d2 << "]" << endl;
}

void print_basic_waveforms(){
  string names[] = { "sine", "triangle", "sawtooth", "square" };
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 8; j++) {
      print_to_file(basic_luts[i][j], LUT_SIZE, "..\\basic waveforms\\" + names[i] +
    "\\" + to_string(j) + ".txt");
    }
  }
}


void run_tests() {

  int N = 44100;
  float* wn = new float[N];
  generate_wn(wn, N);

  float* chirp_freq = new float[N];
  generate_linear_chirp_frequencies(chirp_freq, N, DIGITAL_FREQ_20HZ,
    DIGITAL_FREQ_20KHZ*0.5f); //20 to 10k

  float* out = new float[N];
  float* out2 = new float[N];
  float* adsr = new float[N];

  //FILTERS
  cout << "______________________" << endl;
  cout << "STARTING FILTER TESTS" << endl;
  cout << "______________________" << endl;

  cout << "HP24" << endl;
  print_to_file(wn, N, "..//testfiles//filter//white.txt");
  test_filter(wn, out, adsr, N, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f / FS,
     5.0f, &iir_calc_hp24_coeff);
  print_to_file(out, N, "..//testfiles//filter//hp24.txt");
  cout << "LP24" << endl;
  test_filter(wn, out, adsr, N, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f / FS,
     5.0f, &iir_calc_lp24_coeff);
  print_to_file(out, N, "..//testfiles//filter//lp24.txt");
  cout << "BP12" << endl;
  test_filter(wn, out, adsr, N, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f / FS,
     5.0f, &iir_calc_bp12_coeff);
  print_to_file(out, N, "..//testfiles//filter//bp12.txt");
  cout << "HP12" << endl;
  test_filter(wn, out, adsr, 44100, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f /
     FS, 5.0f, &iir_calc_hp12_coeff);
  print_to_file(out, N, "..//testfiles//filter//hp12.txt");
  cout << "LP12" << endl;
```

```
test_filter(wn, out, adsr, 44100, 0.25f, 0.1f, 0.5f, 0.2f, 1000.0f / FS, 20000.0f / FS, 5.0f, &iir_calc_lp12_coeff);
print_to_file(out, N, "..//testfiles//filter//lp12.txt");
cout << "WRITING ENVELOPE" << endl;
print_to_file(adsr, N, "..//testfiles//filter//adsr.txt");
int i = 0;

cout << endl;
cout << "_____________________" << endl;
cout << "STARTING WAVETABLE TESTS" << endl;
cout << "_____________________" << endl;
//WAVETABLE
for (float p = 0.0; p < 4.0f; p += 0.5f)
{
  cout << "POS = " << p << endl;
  test_wavetable(out, N, p);
  print_to_file(out, N, "..//testfiles//wavetable//" + to_string(i) + ".txt");
  i++;
}

cout << "SQUARE CHIRP" << endl;
test_wavetable_sweep(out, N, chirp_freq, 3.0f);
print_to_file(out, N, "..//testfiles//wavetable//square chirp.txt");

cout <<  endl;
cout << "_____________________" << endl;
cout << "STARTING GENERATOR TESTS" << endl;
cout << "_____________________" << endl;
//GENERATOR
cout << "VIBRATO" << endl;
test_vibrato(out, N);
print_to_file(out, N, "..//testfiles//generator//vibrato.txt");

float w[] = { 0.0f, 0.33f, 0.67f, 1.0f };

for (int i = 0; i < 4; i++)
{
  cout << "WIDTH = " << w[i] << endl;
  test_stereo_width(out, out2, N, w[i]);
  print_to_file(out, N, "..//testfiles//generator//width L " + to_string(i) + ".txt");
  print_to_file(out2, N, "..//testfiles//generator//width R " + to_string(i) + ".txt");
}

cout << endl;
cout << "_____________________" << endl;
cout << "STARTING WAVESHAPER TESTS" << endl;
cout << "_____________________" << endl;


//WAVESHAPE
for (int i = 0; i < 5; i++)
{
  float ip1 = (float)i + 1.0f;
  float g_sin = PI / 2.0f * ((float)i + 1.0f);
  float g_tanh = ip1 * ip1 * 2.0f;

  cout << "G_TANH = " << g_tanh << " G_SIN = " << g_sin << endl;

  test_waveshape(out, out2, N, chirp_freq, g_tanh, g_sin, true);
  print_to_file(out, N, "..//testfiles//waveshape//tanh " + to_string(i) + ".txt");
  print_to_file(out2, N, "..//testfiles//waveshape//sin " + to_string(i) + ".txt");

  test_waveshape(out, out2, N, chirp_freq, g_tanh, g_sin, false);
  print_to_file(out, N, "..//testfiles//waveshape//tanh " + to_string(i) + " no aa.txt");
  print_to_file(out2, N, "..//testfiles//waveshape//sin " + to_string(i) + " no aa.txt");
}

cout << endl;
cout << "_____________________" << endl;
cout << "STARTING ADSR TESTS" << endl;
cout << "_____________________" << endl;
cout << "ADSR WITH RETRIGGER" << endl;
test_adsr(out, N);
print_to_file(out, N, "..//testfiles//adsr//adsr retrigger.txt");

cout << endl;
cout << "_____________________" << endl;
cout << "STARTING MANAGER TESTS" << endl;
cout << "_____________________" << endl;
cout << "TESTING MANAGER OPERATION" << endl;
test_gm_operation(out, N);
print_to_file(out, N, "..//testfiles//manager//operation.txt");

cout << "TESTING MANAGER OVERLOAD" << endl;
test_gm_overload(out, N);
print_to_file(out, N, "..//testfiles//manager//overload.txt");

}
```

Listing E.1: Code used to generate test data for chapter 4

Appendix F

Project Planning Schedule

This is an appendix.

Appendix G

Outcomes Compliance

This is another appendix.