STUDENT REPORT
AALBORG UNIVERSITY

Neptune

An Intermediate Language for Music Composition on the Commodore 64

Emil S. Andersen, Felix B. Lindberg
Alberte Lohse, Nikolaj K. van Gool
Cecilie S. Vebner

AALBORG UNIVERSITY
STUDENT REPORT

Institut for Datalogi
AC Meyers Vænge 15
DK-2450 København

Title
Neptune

Project period
February - May 2025

Group
Group 8

Group members
Emil S. Andersen
Felix B. Lindberg
Alberte Lohse
Nikolaj K. van Gool
Cecilie S. Vebner

Semester
SW4

Supervisor
Levs Gondelman

Number of pages

Abstract

Contents

Chapter 1

Introduction

Most people are familiar with the distinctive sound of chiptune music. This musical style is unique as it is not performed by traditional instruments, but is instead artificially synthesised by sound chips of gaming consoles and computers from the late 1970s to early 1980s [1]. It evokes a sense of nostalgia, reminiscent of a simpler time; the golden age of video arcade games.

Chiptune artists had to work within the limitations of the hardware, both in terms of the sounds it could produce, but also the technical knowledge required to program it.

As technology has advanced through the years, the creation of chiptune music is no longer strictly confined to retro hardware, as it is now possible to replicate with the use of modern software. With this new technology, musicians began producing music which emulated the nostalgic sounds of early video game music, but without the constraints of the hardware [2].

While chiptune as a genre has evolved beyond the need for retro hardware, the novelty of authentic chiptune music has stayed strong over the years and even had a resurgence in recent times.

In this project, we aim to pay homage to the early years of chiptune music. Specifically, we want to focus on a computer with a well-earned reputation within the chiptune community: The Commodore 64.

The Commodore 64, often referred to as C64, was known as a machine with a synth as its heart. This made it an obvious choice for those interested in writing digital music when it debuted in 1982 [1, p. 85]. Its sound chip, the SID 6581, was popular even after newer, more sophisticated chips had been released due to the charm of its distinctive grit and texture [1, p. 79]. It has limited capabilities by modern standards, but these limitations provide a creative challenge to anyone skilled enough to master it.

As stated by distinguished composer Kenneth B. McAlpine, "(...) writing music on the Commodore required a high degree of musicality and a deep understanding of the low-level hardware architecture" [1, p. 69]. Translating musical scores into very specific low-level instructions becomes its own specialised skill. Even in modern times with all the knowledge which is readily available within this area, there is no clear guide which can encompass every skill required to produce even a simple melody on the C64.

We believe that it should be possible for anyone with an interest in writing music to do so for older hardware, and aim to do our part to bring this experience to a wider audience.

This bar of entry could be lowered substantially using modern technology, as our target group could use a higher level programming language, an abstraction of the usual BASIC syntax, which is more intrinsically tied to music theory.

We have achieved (...)

In this project, we aim to bridge the gap of knowledge and make the world of authentic chiptune music more readily available. We aim to give musicians the opportunity to express themselves without the steep learning curve of low-level programming and hardware architecture.

Chapter 2

Language Design

In this chapter we will expand on the general considerations leading to the syntax of Neptune. The source language will then be formalised by specifying the syntax and semantics. Finally the language will be evaluated by exploring it's readability, writability, and reliability.

2.1 Designing the Source Language

The first decision to be made when designing a language is which programming paradigm is most fitting for the context of the language. The two primary paradigms, imperative and declarative, each constitute a distinct model for problem-solving with rules and principles. Where an imperative programming language will describe sequences of instructions to accomplish a task, a declarative programming language describes the tasks that must be accomplished. Therefore, declarative programming focuses more on the end result than the computations that produce the result [3].

In the case of music, it is rarely defined by variables or computations with control structures, but instead with notation of predefined notes, like sheet music. For this reason, our source language is purely declarative, as users can only specify what music should be played, rather than controlling the procedures for producing the music [4, p. 714].

Notes

As the aim is to create a language for writing chip tune music for the Commodore 64, taking inspiration from the music theory upon which chip tune music is based is obvious, as it could make the syntax more intuitive for people with musical knowledge to comprehend and build upon. The manual for the Commodore 64 shows that the SID chip follows Western music theory, and therefore it is most natural that our syntax does as well. For inspiration, we explored existing solutions for compiling music with a programming language. Among other tools, the music engraving tool Lilypond came up, which compiles programs into sheet music following Western music theory and notation [5].

Figure 2.1 shows the LilyPond syntax that inspired our design. The syntax consists of a list of notes represented by a tone name and a fractional duration. The tones are represented by one of the letter names: c, d, e, f, g, a, and b [6]. In Western music theory, there are 12 distinct tones, so accidentals are used for the remaining 5 notes. Accidentals are symbols, such as sharps (♯) and flats (♭), that indicate that a tone should be raised or lowered [7, p. 7]. In the LilyPond syntax in figure 2.1, the sharp accidental is represented with the letter *s*. The duration of a note is indicated by the number following the tone name. In our language, the fractional duration is limited to the range of 1 to 16 for the sake of simplicity (see Appendix B.2 for an explanation of the numbers) [7, p. XIV].

```
    g8 fs a4
```

```
    g:8:3 f#:8:3 a:4:3
```

Figure 2.1: At the top, the LilyPond syntax for a sequence of notes, with the corresponding syntax in Neptune shown below

In LilyPond's syntax, if a note lacks a duration, it inherits the previous note's duration. To simplify implementation and enhance syntax comprehensibility, a standardised way to represent each note explicitly is used in our language. As we are designing an intermediate language, this standardised note structure should make it straightforward for external developers to build upon. The structure will be that each note will be written with a tone letter name, an optional accidental, a duration, and an octave. The octave represents how high or low the pitch of the tone is, and octaves are denoted with numbers (see Appendix B.5). As mentioned, Western music has twelve distinct notes, though a piano has more than twelve keys. This is because the pattern of the twelve tones repeats up and down the piano, and the octave represents how low or high on the piano a tone should be played. Consequently, A4 has a higher pitch than A3. Therefore, providing control over the octave is crucial for creating varied and interesting melodies. LilyPond's symbol for rest notes, which are pauses in the music is also used in our syntax. These notes are represented by the letter *r*, followed only by a duration. For accidentals, the ♯ symbol is used for raising a note, following traditional music notation, and an underscore _ for lowering a note, instead of the *b* symbol to avoid syntax conflicts.

Sequences

The next step is to define the grouping of these notes, which enables the writing of melodies. To visually group notes together, notes are written in sequences, using curly brackets as delimiters. Similar to LilyPond, whitespace is used to separate notes within sequences, as this allows users to format the sequences however they prefer, which can improve comprehensibility. Users also have the possibility to write the **|** symbol in the sequences, as it is reminiscent of a separator used in Western music notation. Furthermore, sequences are assigned identifiers to easily manage and reuse them without repeatedly defining the entire sequence. To offer flexibility, sequences can be any length, as this supports defining an entire song within a single sequence or multiple shorter sequences that can be combined. The assignment operator '=' is used to visually assign sequences to identifiers for better comprehensibility, and as is very common in most programming languages. This symbol is used in the source language whenever assignments are made. An example of concrete syntax of the source language, which defines sequences of notes, can be seen in Figure 2.2.

```
sequence seq1 = { c:1:4 }
sequence seq2 = { d:4:4 e_:4:4 }
sequence seq3 = { g:8:3 f#:8:3 a:4:3 }

```

Figure 2.2: An example of the source language syntax for defining a sequence of notes

Musical parameters

As mentioned previously, the fractional duration of each note is denoted by a number. However, this number alone cannot give a duration, as the duration is relative to the tempo, being the number of beats per minute and the time signature (see Appendix B.2 and B.4) [8]. Therefore, it is important to add syntax that allows control over the musical parameters: tempo, time signature, and standard pitch. The pitch of every note is calculated based on the frequency of the note A4, so by adjusting the standard pitch, you can raise or lower every note [9]. For an example of the concrete syntax, see Figure 2.3.

Voices

Finally, there should be structures that allow sequences to be combined into whole sections of music. Among many possible designs, the most straightforward solution is to take inspiration from the voice system of the SID chip (see section 3.1). The sound played by the SID chip consists of three voices, which the syntax will reflect, with each voice being represented as an array of sequences. Moreover, the use of different waveforms is central to producing chiptune music, as it allows for recreating the sounds of different instruments. Noise, triangle, sawtooth, and pulse are the four waveforms that the SID chip supports. Consequently, for each sequence, a waveform can be specified, making each element of the voice array a pair of a sequence and its associated waveform. Together with the flexible length of sequences, this allows users to change the waveform for every note if a sequence consists of a single note, which is consistent with the functionality of the SID chip. Figure 2.3 shows a complete Neptune program, including the syntax for voices . For a more detailed explanation of sound and waveforms, see Appendix D.

```
        tempo = 120
timeSignature = (4,4)
standardPitch = 440

sequence seq1 = { c:1:4 }
sequence seq2 = { d:4:4 e_:4:4 }
sequence seq3 = { g:8:3 f#:8:3 a:4:3 }


voice1 = [(seq1, square)]
voice2 = [(seq2, triangle), (seq2, triangle)]
voice3 = [(seq3, sawtooth), (seq2, noise)]

```

Figure 2.3: An example of a complete program written in Neptune, with syntax for defining the musical parameters, sequences, and voices

2.2 The Formal Syntax and Semantics of Neptune

When formalising a programming language, it is important to cover the *syntax* and *semantics*. The syntax of a programming language is the rules that define the structure and arrangement of its statements and expressions. Semantics is the meaning of those statements and expressions, and how they are interpreted [4, p. 134]

2.2.1 Syntax

The grammar of a language specifies permissible arrangements of words with a set of production rules. We will in this section use grammar and CFG interchangeably to refer to the grammar of our language.

We use EBNF (Extended Backus-Naur Form), a formal notation for specifying programming language syntax, to define the grammar of our language. Aligned with EBNF notation, abstractions denoted with <> are nonterminals. These nonterminals are defined on the left-hand side. They can have multiple distinct definitions, representing various possible syntactic structures. On the right-hand side are the terminals. Terminals consist of lexemes[1] and tokens [4, p. 138]. It should be noted that in our CFG, the production rules are denoted with ::= to distinguish between the left-hand side and right-hand side. The assignment operator '=' is part of the language syntax and is simply a terminal in the grammar.

[1]Lexemes are the lowest levels of syntactic units. A lexeme is a sequence of characters in the source language that matches a pattern defined by a token.

$$
\begin{aligned}
\langle \textit{file} \rangle &::= \\
&\quad \text{tempo} = \langle \textit{int} \rangle \\
&\quad \text{timeSignature} = (\langle \textit{int} \rangle, \langle \textit{int} \rangle) \\
&\quad \text{standardPitch} = \langle \textit{int} \rangle \\
&\quad \langle \textit{seqdef} \rangle^{*} \\
&\quad \text{voice1} = [(\langle \textit{ident} \rangle, \langle \textit{waveform} \rangle)^{*}] \\
&\quad \text{voice2} = [(\langle \textit{ident} \rangle, \langle \textit{waveform} \rangle)^{*}] \\
&\quad \text{voice3} = [(\langle \textit{ident} \rangle, \langle \textit{waveform} \rangle)^{*}] \\
\langle \textit{seqdef} \rangle &::= \text{sequence } \langle \textit{ident} \rangle = \langle \textit{seq} \rangle \\
\langle \textit{seq} \rangle &::= \{\langle \textit{note} \rangle^{+}\} \\
\langle \textit{note} \rangle &::= \langle \textit{tone} \rangle\ \langle \textit{acc} \rangle?\ :\ \langle \textit{frac} \rangle\ :?\ \langle \textit{oct} \rangle? \\
\langle \textit{tone} \rangle &::= a \mid b \mid c \mid d \mid e \mid f \mid g \mid r \\
\langle \textit{acc} \rangle &::= \# \mid \_ \\
\langle \textit{frac} \rangle &::= 1 \mid 2 \mid 4 \mid 8 \mid 16 \\
\langle \textit{oct} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \\
\langle \textit{waveform} \rangle &::= \text{noise} \mid \text{vPulse} \mid \text{sawtooth} \mid \text{triangle} \\
\langle \textit{ident} \rangle &::= \langle \textit{string} \rangle \\
\langle \textit{string} \rangle &::= [\text{a-z A-Z}]^{+} \\
\langle \textit{int} \rangle &::= [0\text{-}9]^{+}
\end{aligned}
$$

The CFG illustrates that our language is *syntactically strict* as it specifies the valid structure and order of tokens. Any deviations from these rules will not be tolerated, as this strictness will be enforced during parsing (this will be expanded on further in section 4.4).

2.2.2 Semantics

Semantics is the study of meaning in relation to the behaviour of a programming language [10, p. 3].

Mainstream programming languages are usually defined by operational semantics, which is a formal way of defining the meaning of a programming language by its computational steps. However, in our case, operational semantics make little sense since our language does not perform any actual computation, it only processes predefined sequences of declared notes. The meaning of our language is music, or to be more precise, sound waves. This invites us to consider a different approach to formally describing the semantics of Neptune, namely denotational semantics, where the mathematical functions mapped to our programming language constructs are defined [4, p. 161].

The semantics of our language is can be denoted by the following function:

$[\![\cdot]\!] : \textit{neptune} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$

This is a function which maps our language to a function $\mathbb{R} \rightarrow \mathbb{R}$ which synthesises a sound wave.

We will not delve further into the precise denotational definitions of our semantics, since this is outside the scope of our project. Instead, we will provide some intuition of how sound waves are synthesised based on our syntax.

The shape of the synthesised sound wave - its waveform - depends on what is declared in the source language; *noise*, *vPulse*, *sawtooth* or *triangle*. A full overview of how waveforms are synthesised, as well as graphical representations of each waveform, can be found in Appendix D.4.

Each note in our language denotes a sound wave in time. The frequency of this sound wave is calculated from its tone, accidental and octave with a basis in the standard pitch, which we will further explore in section 4.5. The resulting frequency of the sound wave can be seen in the density of iterations in the sound wave, as described in Appendix D.2.

The duration of the sound wave for each note is calculated from its fractional duration in the context of both the tempo and time signature. A more thorough explanation is shown in section 4.5. This duration determines the horizontal length of the sound wave.

Sequences are merely concatenations of the sound waves synthesized by individual notes, and each voice is a concatenation of the sequence sound waves they contain. This leaves us with three overlapping sound waves. These can be thought of as separate, or combined into one complex sound wave such as with additive synthesis (see Appendix D.4).

2.3 Language Evaluation Criteria

Programming languages cannot be strictly evaluated as simply good or bad. The context of the language must be taken into account. What is the purpose of the language? Which problems does it set out to solve? And who are the end users?

As previously discussed, our language will be tailored to a narrow problem domain, namely music composition, and intends to facilitate interaction with the SID chip at an intermediate level of abstraction. With this in mind, we will explore the three main language evaluation criteria: *Readability*, *writability*, and *reliability* [4, p. 31]. These three metrics provide us with a more specific terminology to evaluate our design decisions.

2.3.1 Readability

Readability refers to the ease of which users can comprehend and build upon a program written in a specific language. This is especially important when it comes to maintenance and extension of both programs written in the language as well as the language itself.

As Neptune is intended to be an intermediate language, readability is crucial. We aim for a high degree of simplicity, which relates to these three points:

- **Overall simplicity:** The syntax will have a limited scope.
- **Sparse feature multiplicity:** Little to no variation in the way different features can be implemented.

- **No operator overloading:** The purpose of each operator is clear and distinct.

This is all possible due to the declarative nature of our language, as we will not have any meaningful control flow. We aim to support the functionality of the SID chip, not to add any clever functionality or shortcuts. The cost of this high degree of simplicity is an obtuse language which is tedious to write programs for. That is not an issue in our specific case, as our language is not meant to be used as is. We are leaving room for any person to extend the language to fit their specific use case, which means that a more generalised and simplistic approach is fully appropriate.

On the other hand, the orthogonality[2] of our language is scant. Anything resembling a data type has its own specific rules and limitations. Parameters are isolated in the header of the file and used only in calculations within the compilation process. It is also not possible for users to add custom parameters or define any of their usages. Sequences are defined with their own distinct syntax, and only intersect with voices in that their names are used to play them in the desired order. The effect of this is that the language is very inflexible. This is not an issue in the context of our language. We leave space for extensions, which could branch into many directions and introduce different degrees of flexibility.

In alignment with the previous points, we have a list of reserved keywords.

Keyword	**Category**	**Syntax Example**
`tempo`	Parameter	`tempo = 300`
`timeSignature`	Parameter	`timeSignature = (4,4)`
`standardPitch`	Parameter	`standardPitch = 440`
`sequence`	Sequence	`sequence seq1 = {c:4:5 d:4:5 e:2:5}`
`voice1`	Voice	`voice1 = [(seq1, vPulse), (seq1, sawtooth)]`
`voice2`	Voice	`voice2 = [(seq2, vPulse), (seq2, sawtooth)]`
`voice3`	Voice	`voice3 = [(seq3, vPulse), (seq3, sawtooth)]`
`vPulse`	Waveform	`voice1 = [(seq1, vPulse)]`
`sawtooth`	Waveform	`voice1 =[(seq1, sawtooth)]`
`triangle`	Waveform	`voice2 = [(seq1, triangle)]`
`noise`	Waveform	`voice3 = [(seq1, noise)]`
`a,b,c,d,e,f,g`	Note	`sequence seq1 = {c:4:5 d:4:5 e:2:5}`
`r`	Rest	`sequence seq1 = {c:4:5 r:4 e:2:5}`

Table 2.1: Keywords in our source language

Each keyword has a distinct purpose in that the meaning behind the syntax remains the same regardless of the context it is placed in. The context which different syntactic structures are allowed in is, in fact, highly restricted. For instance, a note cannot be directly declared in a voice, and sequence names cannot take the form of any keyword.

2.3.2 Writability

Writability, in contrast to readability, concerns the ease of which users can write programs in a specific language. Though these two concepts are strongly related, there is a difference in the matter of

[2]A language is orthogonal if a relatively small set of its primitives can be combined to build control and data structures. Crucially, every combination of primitives must be *legal* and *meaningful*, implying a consistent set of rules for how data types can be used [4, p. 33]

perspective. As the previous section hints at, writability is not prioritised as highly in our endeavour.

The expressivity of our language is very limited. Writing more complex pieces of music would require many lines of dense code, since there are no shortcuts for specific purposes. This is fine, since the idea is not that music will be written directly at this level.

Extending our language does require an understanding of its syntax, but only for the purpose of generating it from a more efficient and expressive language. However, no matter how the language is extended, it is eventually forced to comply with the strictness of our language. This increases the likelihood that input accepted by our compiler will be compatible with the C64 and produce some sound output.

2.3.3 Reliability

Reliability refers to a language being consistently functional in all relevant use cases. Reliability is also closely tied to readability and writability, as a language which is easier to read and write is usually also more reliable [4, p. 39].

There are some additional areas in which reliability can be measured and evaluated:

- **Type checking:** Since our language has no types in the traditional sense, static errors will be related to lexing and parsing rather than typing.
- **Error handling:** Since program execution is non-alterable during runtime, we will not focus on handling dynamic errors.
- **Aliasing:** We will have no aliasing, as sequences cannot point to other sequences, they must be defined as separate entities. Only one name is mapped to each sequence.

Chapter 3

Target Language

Having established our source language, the next step is to identify our target language. Before we determine how to transition from the source to the target language, we must first have a clear understanding of what we aim to achieve. Our objective is to create a program that generates sound on the Commodore 64, which necessitates a thorough understanding of how to communicate with the Commodore 64's sound chip, the SID 6581. We must therefore get proper insight into the functionality of the hardware of the sound chip and its sound-generating capabilities. In this chapter, we will expand on the most necessary functionalities of the SID 6581 (greater detail of the hardware can be found in Appendix C). Furthermore, we ought to comprehend how to communicate with the SID chip, as our chosen target language will shape our interaction with it.

3.1 SID 6581

The SID 6581 is a single-chip music generator consisting of three independent voices. These voices can be executed individually or simultaneously to produce compound sounds. The sound chip provides a wide frequency range, high-resolution control of pitch, tone colour, and dynamics [11].

The block diagram in Figure 3.1 shows the internal structure of the SID chip. Each voice has a tone oscillator, an envelope generator, and an amplitude modulator. The tone oscillator is responsible for generating a sound wave with the pitch and waveform specified within its registers. The amplitude modulator controls the volume dynamics of each note, based on four values within the envelope generator: Attack, Decay, Sustain, and Release (ADSR). Filters and volume is also controlled globally across all three voices [11].

These terms will be expanded on in the following subsections.

3.1.1 Voices

Based on our understanding of the workings of the SID 6581, we define a voice as follows: a voice is a sound signal transmitted from one place to another, and each voice generates and manipulates a single sound wave.

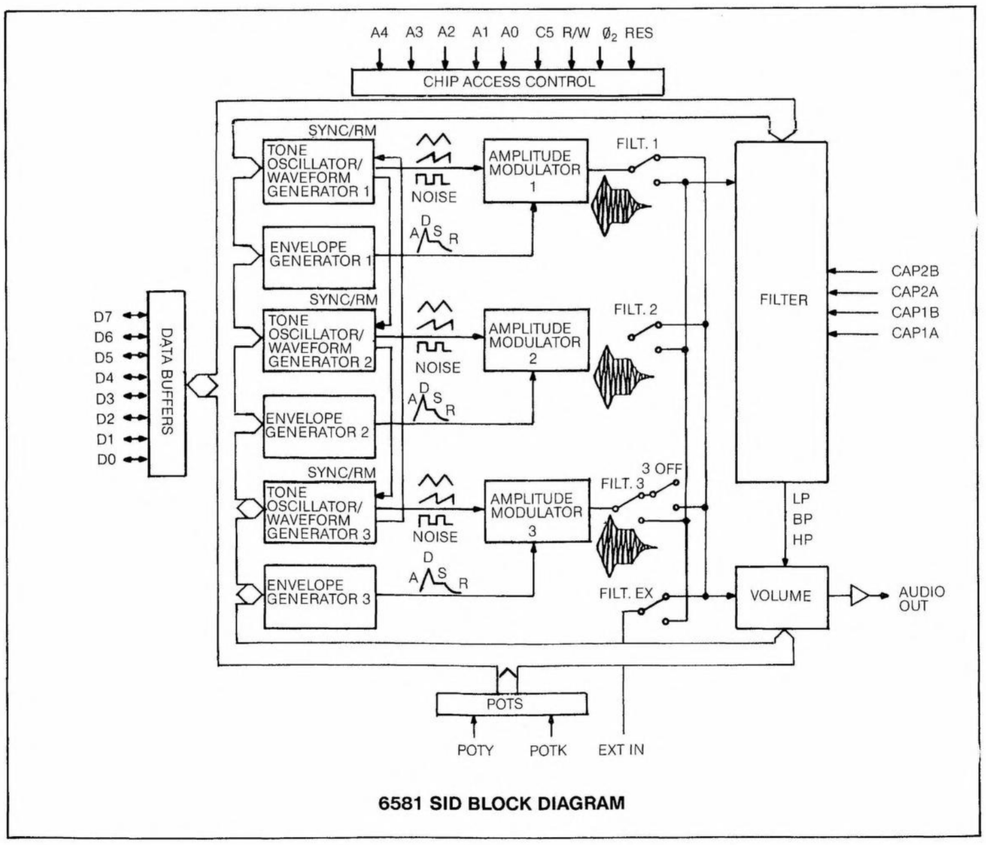


Figure 3.1: Block diagram of the SID 6581 sound chip [11]

In the SID 6581, the individual voices operate independently of one another, but can create a polyphonic[1] sound if executed in conjunction with one another. The sound output of a voice is produced with different waveforms: triangle, sawtooth, variable pulse or noise. A feature of the SID 6581 is that it is possible to change the waveform of a voice to a different waveform (see figure 3.1.5) [11].

3.1.2 Oscillators

The three voices each have an oscillator, an envelope generator, and an amplitude modulator. An oscillator is a circuit that generates continuous alternating waveforms. Each oscillator has a range of 0 to 4 kHz[2] [11]. It does so by digitally creating one of the four waveforms and controlling the pitch with multiple frequency parameters given to the chip [13].

A waveform is a simple representation of a sound wave, typically visualised with a graph. In figure 3.1.2, such a graph can be seen, depicting the sound waves' displacement over time, of the SID chip's four waveforms.

- The **triangle** waveform is low in harmonics and has a mellow and flute-like sound.

[1]"polyphony, in music, the simultaneous combination of two or more tones or melodic lines (the term derives from the Greek word for "many sounds"). Thus, even a single interval made up of two simultaneous tones or a chord of three simultaneous tones is rudimentarily polyphonic."[12]
[2]Kilohertz

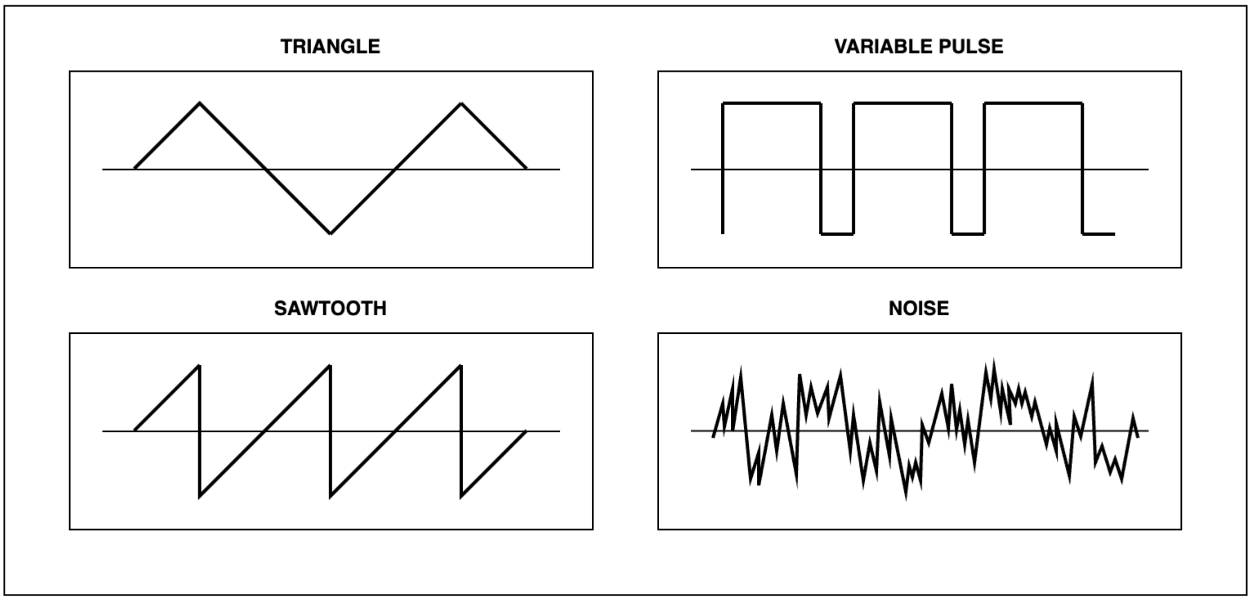


Figure 3.2: The four waveforms of the SID chip: triangle, sawtooth, variable pulse, and noise [13]

- The **sawtooth** waveform contains both even and odd harmonics, giving it a bright and brassy sound.
- The **variable pulse** waveform has a varied sound, fluctuating from a bright and hollow square wave to a nasal and reedy [3] pulse wave. Adjusting the pulse width registers changes the harmonic contents of the pulse waveform, affecting how its tone qualities are produced.
- The **noise** waveform (white noise) is the last waveform of the SID 6581. Its sound quality ranges from a low rumbling to hissing white noise [11].

When it comes to controlling a sound wave's frequency, each oscillator has two 8-bit registers for this purpose, a low byte and a high byte. These 16 bits combined will result in the frequency of the generated sound wave [13]. As an example, to set the frequency of Voice 1, you could set the low frequency to $FB and the high frequency to $04. The combined frequency value would be $04FB (1275 in decimal). This value would then be used by the SID Chip to calculate the frequency at which the Oscillator's waveform should be generated, which corresponds to the tone D#.

3.1.3 Envelopes

An instrument playing a certain note creates a sound whereof volume has a certain shape which changes over time. When pressing a piano key, it in an instant produces a high volume note and drops either over time or almost instantly (dependent on if the sustain pedal is being pressed) to silence. In contrast, a keyboard set to strings typically ramps up volume over time. These ramp up, note sustain, note decay, and release times can be defined as envelopes. Multiple types of envelopes exists such as

[3] A thin and high sound

AHD[4], HADSR[5] and many more, but one of the more basic envelopes which the SID 6581 is able to create is ADSR[6] [14].

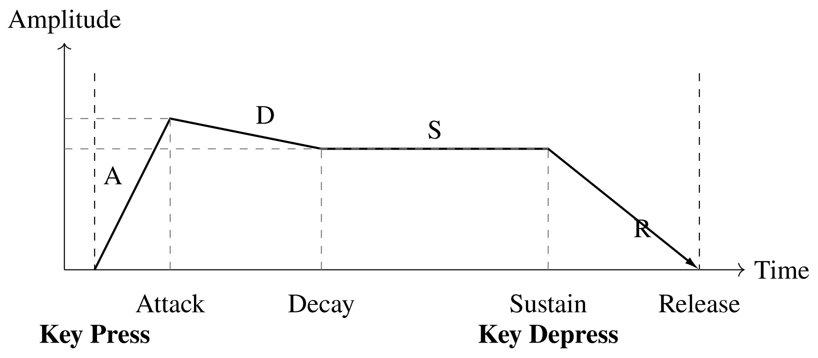


Figure 3.3: ADSR Example

The ADSR Envelope within the SID 6581 is digital and is controlled using 4-bit parameters, which in turn offer 16 steps for each phase.

- **Attack**
  The attack phase is the phase that regulates the time it takes for the sound to rise in amplitude from complete silence. A higher 4-bit value equals slower rise times [11].
- **Decay**
  The decay phase defines the time it takes for a sound to fall in amplitude, from the current wave form's volume to the sustain level. A higher 4-bit value equals slower decays [11].
- **Sustain**
  The sustain level is the amplitude at which the sound is held before being released. The amplitude at which it is being held is controlled with the 4-bit value, which specifies a percentage of the peak volume, so the sustain parameter sets the target amplitude that the decay parameter seeks for [11].
- **Release**
  The release parameter controls the time it takes for the sound to fade out to silence after the note is released. A higher 4-bit value results in a longer release time [11].

As mentioned above, the envelope generators have a 4-bit resolution and changing the parameters is done in steps. This can result in audible changes at slower rates, but contributes to the special sound that the SID 6581 produces [13].

3.1.4 Filter

The filter serves the purpose of generating elaborate and dynamic tone colours. This is done by changing the harmonic structure of a waveform, thereby producing a customised sound output [13]. It does so by removing specific frequencies from the final sounds.

[4]Attack, Hold, Decay
[5]Hold, Attack, Decay, Sustain, Release
[6]Attack, Decay, Sustain, Release

It is a multimode resonant filter, as it consists of three different modes: a low-pass filter (LP), a high-pass filter (HP), and a bandpass filter (BP) [1]. The filter modifies frequency content, allowing for certain frequencies to pass and reducing or eliminating others. The SID 6581's filter has a cut-off frequency ranging between 30 Hz and 10 kHz, approximately [11]. The LP filter allows lower frequencies to pass, rejecting those above the specified cut-off. Rejection is done by attenuation, meaning the strength of the sound will be gradually reduced as the voice is routed through the filter. Adversely, the HP filter will pass the frequencies above the cut-off and attenuate the ones below. The BP filter will pass frequencies within a specified range of the cut-off and attenuate the frequencies outside of the range [1].

As visualised in Figure 3.1, a voice goes through its amplitude modulator. From there, it will either be routed through the filter or bypassed. For the filter to have an effect on the audio output, one of the three filter modes must be selected.

3.1.5 SID Control Registers

Controlling these different functionalities in the SID chip can be done by writing to a range of addresses within the C64's memory, spanning from $D400 to $D418. The full mapping of these registers can be seen in Appendix C.2. To sum up, each voice has its own set of the seven following registers:

- **0-1** Defines the low and high byte of a note's frequency.
- **2-3** Defines the low and high byte of the width of a waveform, but applies only to pulse waveforms.
- **4** Control register, in which each bit controls a different aspect of the sound. For instance, there are bits representing each waveform, as well as a gate bit which can be used to silence the voice altogether.
- **5** Defines the Attack and Decay of the envelope generator, with four bits assigned to each.
- **6** Defines the Sustain and Release of the envelope generator, with four bits assigned to each.

In addition, there are four filter registers which control the filter cutoff, selected filters, and global volume, among other things.

3.2 Choice of Target Language

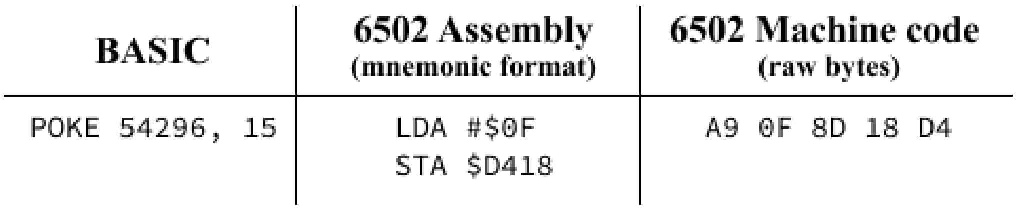
Interacting with the SID chip and programming its behaviour can be done at three levels of abstraction.

At the lowest level, we have machine code. This is simply a series of bytes encoding instructions, values and memory addresses. It has to be in the specific format that the C64's microprocessor expects and is able to interpret. Writing programs in machine code is the most efficient option, at the cost of both readability and writeability.

The second level of abstraction is Assembly 6502, which is simply machine code styled to be human readable [15]. The assembly language introduces mnemonic representations of instruction codes in the form of three-letter abbreviations. Traditionally, this was a method to outline the program structure, which could then be converted into machine code by hand. This is known as *hand assembly*. The process has since been automated by introducing a piece of software called an *assembler*. There are many varieties of assemblers with slight syntax variations and shortcuts, such as labels to represent addresses in memory.

The third and final level of abstraction is BASIC, which is the high level language built into the initial prompt of the C64. This language was designed to be usable by those who are not otherwise technically inclined [16]. The C64 shipped with version 2.0 of BASIC, which was outdated even at the time. It is limited in many aspects, and especially rushed when it comes to sound and graphics [1]. Instructions for creating music in BASIC strongly resemble their lower level counterparts, as users are still required to read and write to addresses in memory.

A comparison of these three abstraction levels can be seen in figure 3.4. It shows how the master volume of the SID chip is set to its maximum value.



BASIC	6502 Assembly (mnemonic format)	6502 Machine code (raw bytes)

`POKE 54296, 15`	`LDA #$0F`
	`STA $D418`
		`A9 0F 8D 18 D4`

Figure 3.4: Setting the SID chip to maximum volume in three levels of abstraction, from BASIC down to machine code.

In BASIC, the keyword POKE is used to write to the memory address 54296, setting its value to 15. In assembly, this instruction is split into two steps, as we cannot move data directly between memory addresses at this level and below [15, p. 11]. First, `LDA` (LoaD Accumulator) loads the hexadecimal equivalent of 15 into the accumulator[7]. Only then can we use `STA` (STore Accumulator) to write the value to the memory address D418, which is the hexadecimal equivalent of 54296. In machine code, instructions are in the form of hexadecimal values. Here, `LDA` is `A9`, and `STA` is `8D`. The order of bytes in the memory address are also in reverse order, since the C64 microprocessor is *little endian* [8] [18].

For our compiler, we have chosen to work in assembly. It is low-level, and thus very efficient, but remains at a decent level of readability. This helps us to more easily evaluate the output of the compiler, with fewer conversions and mnemonic lookups. The marginal increase in readability of BASIC in this context is far outweighed by its slow processing speed.

As mentioned earlier, assembly introduces the need for an assembler. Countless assemblers can be found online, and there are not many resources comparing benefits and drawbacks of them all. The general consensus in online forums is that the choice of assembler should be based on one's preference of certain syntax styles [19]. For our project, we have chosen the assembler *dasm*. It is open source, cross-platform, and supports the 6502 architecture. This means that the assembler works on Windows, Mac and Linux, and the entirety of the source code is freely available on GitHub. In

[7]A register which holds a hexadecimal value. This value can be copied and modified without affecting any memory. It is used as effective temporary storage of values, and can be used to perform arithmetic operations [13].

[8]Little endian means that bytes are processed in reverse order, least significant byte first [17, p. 78]

addition, we find the documentation to be comprehensible. The syntax has a lot of flexibility, such as not being case-sensitive. All of this means we can spend less time figuring out the assembler, allowing us to focus more on compilation between our source language and target language.

3.3 Outlining the Assembly File

Before our compiler can be implemented, it is vital to understand the structure of the assembly file it should to compile to. The aim is to design an assembly language music routine with functionality corresponding to that of our source language.

3.3.1 Preliminaries

The assembly file should, as required by dasm, start with a header defining the processor as well as the origin of our program, which is the exact memory address a BASIC program is defined in. This is due to the fact that running assembly code in the C64 prompt requires it to be called as a subroutine using the keyword *"SYS"*. This is done in our code by constructing a BASIC program as a series of bytes, using the `dc.b`[9] instruction of dasm. The construction of the BASIC program can be seen in Appendix C.3.

3.3.2 Initialisation

The next step is to define initial values for the SID chip. First, the volume is set to the maximum value to ensure sufficient sound output, and filter cut-offs are cleared since they are not used in our current scope. Then, addresses at the beginning of each voice's byte data are assigned to the equivalent pointer variables (see sections 3.3.4 and 3.3.5).

3.3.3 Setting the Raster Routine

Since we are dealing with music, and as such wish to have control of the timing of notes changing over time, we are defining our own custom *raster routine*. The raster routine refers to the refreshing of pixels on the screen, something which happens once every 20 milliseconds, or frame. These refreshes, or *raster IRQs* (Interrupt ReQuests), can be used as a framework for the timing of notes. By first disabling interrupts completely, the IRQ vector can be updated to hold a routine of our choosing, ended by re-enabling interrupts. This routine is simply named *raster* (see Section 3.3.6), and will now be called on every raster IRQ until the C64 is reset, even after the main program finishes its execution.

3.3.4 Music Data

Before we delve into the music routine which will handle playing music, we will first describe how the music itself is written, the *music data*. Music data defines the structure of voices and sequences as blocks of byte data. The hexadecimal value of each byte determines its meaning.

- **\$00-\$F8** High frequency, low frequency or duration of note.

[9]dc.b means "define constant bytes". The comma separated bytes which follow it are written into the memory in the order they are declared.

- **$F9** Noise waveform
- **$FA** Pulse waveform
- **$FB** Sawtooth waveform
- **$FC** Triangle waveform
- **$FD** Jump to address
- **$FE** Enter sequence
- **$FF** Exit sequence

Due to the syntax of our source language, there is a strict pattern which music data will follow depending on whether it is a voice or a sequence.

Sequences

Sequences contain one or more rows of three bytes, each of which represents a note. The notes will be processed in the order of low frequency, high frequency and duration, and are expected to strictly follow this order. All sequences end with the instruction to exit the sequence. In section 4.6, an example of such a sequence is presented.

Voices

Voices are divided into three blocks of byte data, labelled as *voice1*, *voice2* and *voice3*. For each sequence defined in a voice, there will first be a byte representation of the sound wave, then an instruction to enter a sequence, followed by the label of that sequence. All voices end with a jump to the beginning of the voice. An example of voice data declaration can be seen in section 4.6.

3.3.5 Variables

To manage information about the status of a piece of music over time, a set of long term variables that can easily be navigated and updated are needed. These contain information about the current state of each voice, as well as information related to each waveform, referred to as the *instrument*. Just as with music data, there are some labelled rows of byte data which can be referred to during program execution in conjunction with an index.

Voices

Voice variables contain three columns of byte data, one column for each voice. Each row is labelled as the variable it contains. For instance, the variable `v_counter` holds the remaining duration of a note being played in each voice. Accessing the value for a specific voice is possible by using a voice index, which we have dedicated the X register to. The indexes 0, 1 and 2 are added to the address of a variable to access the desired value. A full overview of our voice variables can be seen in Appendix C.4.

As an example, if the index in register X is currently 1, it means that voice2 is currently being processed. To get the remaining duration of the note played in this voice, the following line of assembly code must be used:

```
                lda v_counter,x
```

We use absolute indexed addressing mode[10] to load the address of *v_counter* plus 1, thus loading the value of the second column into the accumulator. The same method is used when writing new values into variables.

Instruments

Instrument variables work similarly to voice variables where each row is labelled by a different variable name, but here, each of the four columns are tied to different waveforms. To navigate the four instruments, we use an instrument index stored in the Y register. Each instrument stores the hexadecimal representation of the waveform, ADSR as well as information about variable pulse specifically. An overview can be found in Appendix C.4. At this stage, there is no way to modify the instrument variables through our source language. This will be an important next step in ensuring our compiler can handle all possible interactions with the SID chip.

3.3.6 Music Routine

We are now left with only our music routine itself, which is a set of interconnected subroutines which updates the values in the SID chip every frame. As the flowchart in figure 3.5 shows, there are four main components to the music routine.

In the outermost layer, we have our raster routine, which is what we have now stored in the IRQ vector. Then, we have the main loop, which is what iterates through each voice and updates the values according to our music data and variables. Within this main loop, there are two possible branch conditions: One for fetching notes and one for initialising notes. Notes must be fetched two frames in advance, which means that when they are meant to be played, the information will be readily available. Fetching in advance also allows us to turn off the gate bit of the previous tone, starting the release cycle and allowing for a smooth transition between notes. We will now provide more detail for each of these four components of the music routine.

Raster

In this outermost part of the music routine, we use a series of `NOP` instructions to alter the timing of notes very slightly. This gives a tighter sound output. We then enter the main loop of our music routine, play one frame of music, and finally acknowledge the raster interrupt. This is an important step, as it ensures the music routine is not called infinitely, but only once every frame.

[10]This mode of addressing uses a base address and an index value to get the effective address [15, p. 77]

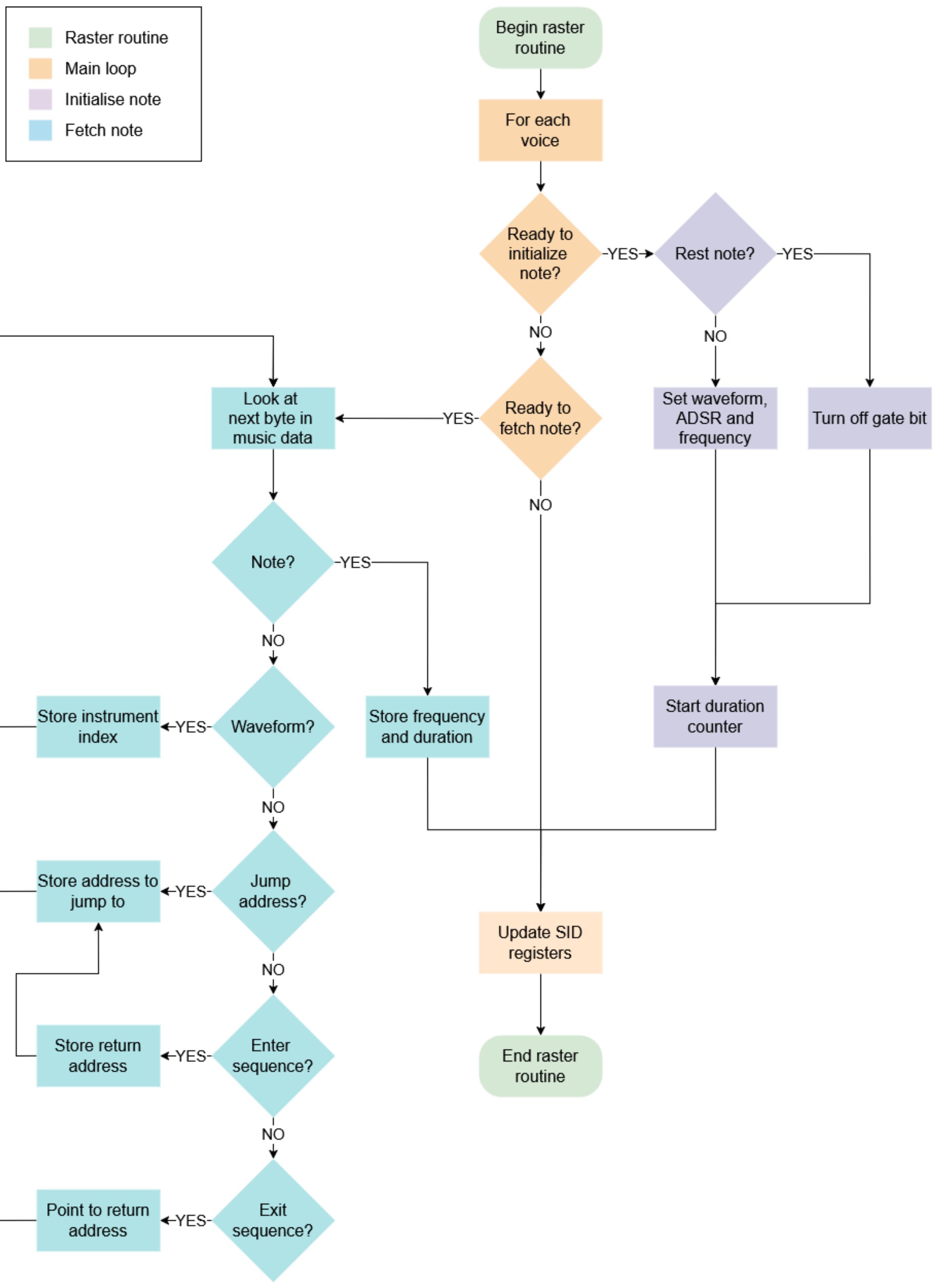


Figure 3.5: This flowchart depicts the overall structure of our music routine. The raster routine is called on each frame, iterates through all three voices in the main loop, and initialises or fetches notes when needed.

Main Loop

As mentioned previously, the main loop processes all three voices in succession. We reserve the X variable as an index, so when voice data or voice variables are accessed, only information relevant to the voice currently being processed is available. For each voice, we first check if any of the two branch conditions are met. When two frames of a note's duration remain, it jumps to the fetch routine. When the duration reaches 0, it jumps to the initialisation routine. Regardless of whether any branches were taken or not, we return to the last part of the main loop which updates the SID chip values: Frequency, waveform and ADSR. We also decrement the counter which tracks the duration of the current note. After all three voices are processed, we return to the raster routine.

Initialising Notes

When this branch is taken, it is because a new note should start playing at this frame. The high frequency of the fetched note is used to check whether or not it is a rest note, since there are no tones with a high or low frequency of 0. Therefore, when the high frequency is zero, we simply turn off the sound output by setting the gate bit to 0. In the case of a note with sound output, we instead store the low and high frequency of the note, as well as waveform and ADSR for the fetched instrument. In both cases, we end the initialisation routine by setting the fetched note duration to the corresponding voice variable.

Fetching Notes

When this branch is taken, it is because a new note is two frames away from being initialised. This step goes through music data, byte by byte, only returning to the main loop when a new note has been found. As explained in section 3.3.4, the value of each byte determines its meaning. If the value of the current byte is $F9 or below, we know that we have found a note. This byte and the following two are expected to be the frequency low byte, frequency high byte, and duration in frames. This information is stored in separate variables, so for instance, the low frequency is not stored in `v_freqlo`, but `v_freqlo_new`. This way, they are readily available at the time of initialisation, not written to the SID chip immediately.

In the case that the current byte is not a note, we must determine which other purpose this byte serves. Values between $F9 and $FC represent a change of waveform. By subtracting $F9 from this byte, we get an index of the instrument we wish to change to. This is stored as the variable `v_instr` to be used in note initialisation.

The remaining values are reserved for the *jump_addr*, *enter_seq* and *exit_seq* routines, respectively. They serve as indicators to jump to the correct routine. In *jump_addr*, we expect the following two bytes to represent an address in the music data we should jump to, which we will then store in the pointer variables. In *enter_seq*, we save the return address we go when the sequence is exited, and then jump to the address of the sequence using the *jump_addr* routine. In *exit_seq*, we simply set the pointer to the previously obtained return address.

In any of the cases where we do not find a note, we process the byte as mentioned above, and then repeat the entire fetch routine with the next byte. The fetch routine will not terminate until a note is fetched.

Chapter 4

Compiler Design

This chapter will expand on the implementation of our compiler. The entire compilation process is explored, from the lexical analysis of the source language to the generation of assembly code, and finally, the execution of the resulting executable. The chapter will cover the design decisions and the rationale behind our implementation, supported by examples and code snippets.

4.1 Specifications

The specifications of our compiler define what we expect our compiler to do. We have used the MoSCoW model to determine the requirements of our compiler, which has been documented in Appendix E. The most significant feature of our compiler is that it must be able to translate our source language into assembly code (see section 3.2). Additionally, the assembly code output by the compiler must be able to interact with the SID chip's control registers, which are in charge of determining the waveform in each voice. Furthermore, the compiler should integrate an external assembler responsible for converting the assembly code into machine code.

4.2 Compiler Architecture

The compiler we have implemented is responsible for reading and converting our source language into assembly code, and an overview of the process of compiling our source language can be seen in Figure 4.1.

Our compiler consists of various phases, which can be divided into two main parts, the frontend and backend. The compiler's frontend incorporates the lexer, parser, and translator. The first phase in the compilation process is where the lexer performs a lexical analysis, a process of deconstructing our source language into tokens. In the following phase, the parser generates an intermediate representation (IR), a representation of the tokens of the source language, in the form of an abstract syntax tree (AST). The generated AST, which we call the source AST, is a high-level IR as it is close to our source language, and it is the first of two ASTs in our compiler. The next phase, the translation, is when the initial source AST is translated into another AST, which we call the target AST, as it is a low-level IR that resembles our low-level target language, assembly code. Through these last two

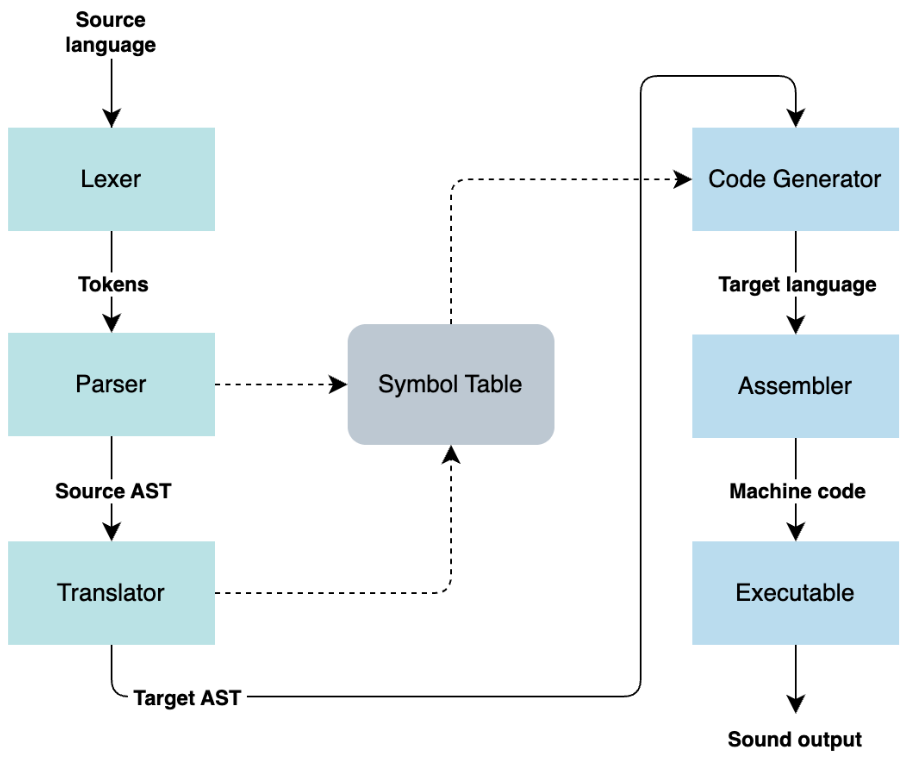


Figure 4.1: The diagram is a visualisation of our compiler architecture. It shows the different phases of the compiler, split into frontend and backend, as well as a symbol table used for storing sequence information.

phases, a symbol table is used to store the sequences. These are the phases and the main processes in the frontend of our compiler.

The backend's responsibility is to convert the target AST into low-level machine code to be executed by the Commodore 64. The first phase of the backend is the generation of an output file, written in assembly, based on the target AST and symbol table, containing music data for the sequences and voices. In the next phase, an external assembler is used to assemble our target language into an executable file, written in machine code, that can be executed on the C64. The assembler is integrated into the compiler, allowing end-users to compile our source language directly into an executable file with a single command in the command line. The machine code can then be executed on an external C64 emulator, producing a sound output authentic to the hardware of the SID 6581.

We have created our compiler using OCaml, a language that was covered in our educational course material, which offers great tools for implementing lexers and parsers. Moreover, OCaml's pattern matching facilitates clear and concise case analyses, making vertical extensions[1] of the compiler straightforward, compared to object-oriented languages. In OCaml, vertical extensions can be

[1]Vertical extensions mean adding or updating features in a system [20]

developed in a single function, whereas in object-oriented languages, changes will often require updating multiple classes [21] [20].

A typical compiler design includes a type-checking as well as an optimisation phase in the backend, however, these phases have been skipped in our compiler design. The declarative nature of our source language means that there are no functions or computations to optimise or explicit data types that require checking.

4.3 Lexer

The lexer is the first phase of the compiler; this is where the lexical analysis happens, which is the first step of the syntactic analysis. In this phase, the lexer takes the source language as input and produces a stream of tokens as output, which will then be handled by the parser in the next step of the syntactic analysis.

To create the lexer, we have implemented OCaml's standard tool for lexical analysis, *ocamllex*. This lexer follows a declarative lexing approach by utilising regular expressions to define legal tokens [22] [4]. During compilation, the source language appears to the compiler as a single string of characters. Therefore, the lexical analyser is responsible for categorising the characters into different types of tokens based on some predefined rules [23]. It performs the lexical analysis by matching the lexemes (input character strings) to the regular expression patterns and performs the semantic action paired with the regex in the lexing rules.

- A *regular expression* is a formal notation that describes a variety of tokens required by programming languages. It is essentially a sequence of characters that defines a search pattern [23].
- A *semantic action* informs the lexer of what to do when it matches a pattern. A semantic action typically instructs the lexer to return a token and sometimes performs some other action as well.

We have designed our lexer to recognise the following token types: *keywords, identifiers, tones, literals, operators, delimiters, comments,* and *white space*. To differentiate between keywords and identifiers, we implemented a hashtable that stores all reserved keywords, allowing for efficient lookup. When the lexer matches a potential identifier, it uses a helper function to search through the hashtable to decide whether it is an identifier or keyword. This function is called within the semantic action associated with the ident regex (see Appendix F.2).

We chose to implement a separate lexing state for handling comments to avoid comments mistakenly being interpreted as code. When the lexer encounters '/*', it jumps to the separate lexer state that is solely concerned with handling comments. This state ensures that the lexer ignores any input between the comment delimiters '/*' and '*/'. Once the lexer encounters the end-of-comment delimiter, it will return to the general lexing rules.

At the end of the lexing rules, we have a catch-all rule for handling inputs that do not match any of the specified token patterns. With this follows a semantic action wherein an exception is raised. The raised exception calls a helper function which uses the ocamllex module *Lexing* to retrieve the location of the lexeme currently being processed in the lexbuf. This function helps provide precise error handling to the lexer.

4.4 Parser

The parser is the second phase of the compiler and the main phase of the syntactic analysis. It tries to match tokens constructed in the lexical analysis with the production rules from the context-free grammar (see section 2.2.1), and if it matches, the AST (see Figure 4.3) is generated. Additionally, the parser detects syntactical errors, signalling with a position in the source language, and an error message [24]. This is handled in a similar way to the Lexer (see section 4.3). It is important to note that the parser of our language terminates execution of the compiler when an error occurs, and does not show any following errors.

In this project, we have chosen to use *Menhir* as our parser generator instead of writing the parser ourselves, similar to how ocamllex generates our lexer. Menhir takes defined production rules and generates the compiler's actual parser based on these rules. The biggest advantage of Menhir is that its error messages are generally more human-comprehensible, and the parsers that it generates are fully reentrant and can be parameterised in OCaml modules more easily. Menhir parses bottom-up, using the LR(1) parsing method, and is the most powerful parser of all deterministic parsers in practice [25]. It scans the input from left to right, and looks for the right-hand side of the production rules to build the derivation tree.

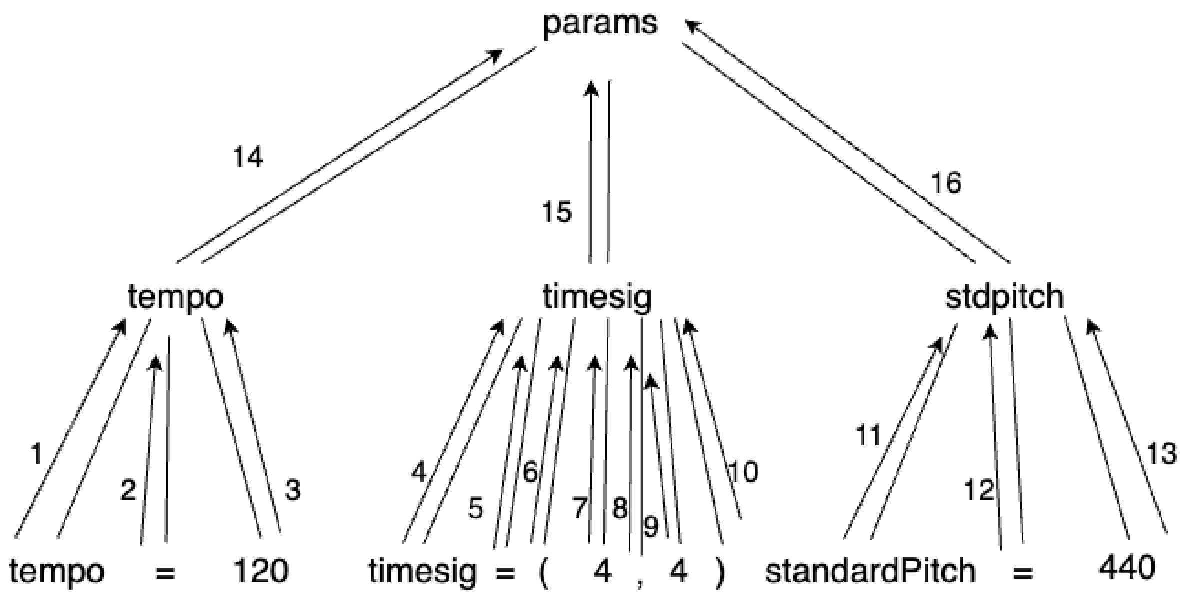


Figure 4.2: A simplified parse tree based on the first three lines of code in appendix H.2. The arrows show how the parser parses from left to right and bottom up

The small parse tree (see Figure 4.2) illustrates how the code is transformed when parsing. It first parses the tempo keyword, followed by the = symbol, the integer 120, and continues in this manner.

See Appendix F.2 for the full parse tree.

The parser receives the tokens generated by the lexer and builds an AST according to Neptune's grammar. An example of this AST building process is the `params` rule (see listing 4.1), which defines how the global parameters are declared and set. The parser expects tokens in this specific order: `TEMPO`, `TIMESIG`, and `STDPITCH`.

```
  params:
| TEMPO ASSIGN t = INT
  TIMESIG ASSIGN SP npm = INT COMMA bnv = INT EP
  STDPITCH ASSIGN sp = INT
```

Listing 4.1: The rule of params

For example, the source code (see listing 4.2) are parsed according to this rule. During parsing, the semantic actions validate and store the tempo, time signature, and standard pitch in the resulting AST. The tokens related to tempo, time signature, and stdpitch are collected into a `params` record, which is then parsed to the `prog` node.

```
tempo = 120
timeSignature = (4,4)
standardPitch = 440
```

Listing 4.2: Source code example for Neptune taken from appendix E

In addition to the producing AST, the parser stores the sequences in a symbol table (see figure 4.1), and an identifier is later used to track the specific sequence in the backend. During parsing, we are also preventing duplication of sequences identifiers. Based on the source code in Appendix H.2, the following AST is generated (see figure 4.3). This AST is a representation of our source language, and holds all information written in the input file. The final AST (see figure 4.4) in return, shows the information which is needed for our target language.

4.5 Translator

The translator uses a series of functions to unpack the source AST generated by the parser and reconstruct its fragments into the equivalent target AST. Some of the translation is rudimentary, such as with voices, where the structure is preserved between ASTs and only the type is changed. The more interesting aspect of the translator is the translation of notes. In the source AST, notes consist of a tone, an accidental, fraction, and octave, which relates to music theory. This format is not compatible with our music routine (see section 3.3.6), which requires a high and low frequency and a duration in frames. Calculating these values now simplifies code generation in the next phase of compilation.

To calculate the high and low frequency of a note, we must first find the frequency output in Hz. In the case of rest notes, both values are simply set to 0. In all other cases, the standard pitch is used as the baseline and then incrementally modified by a number of semitones[2] offsets. These offsets are not

[2]A semitone refers to the distance between each of the twelve tones in Western music theory (see Appendix B.3)

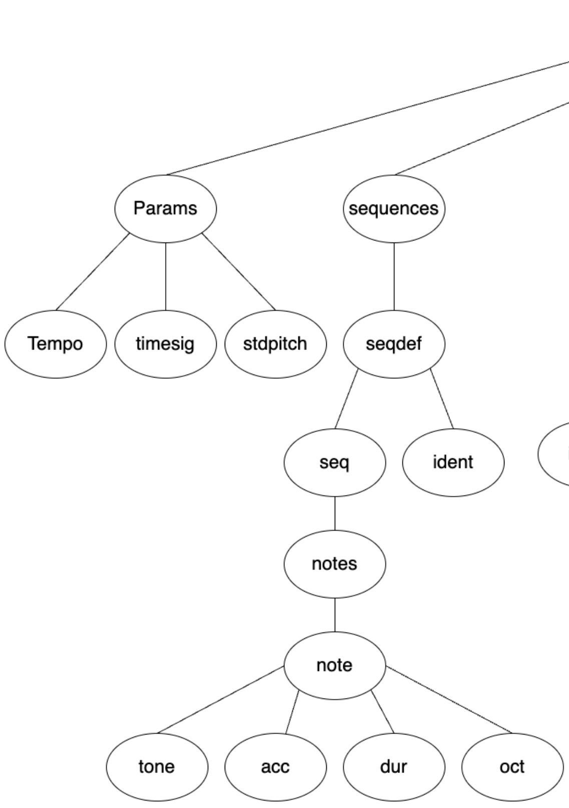


Figure 4.3: A visualisation of the first AST made with our parser

linearly spaced out, but instead follow the ratio $r = \sqrt[12]{2} \approx 1.05946$, where r is one semitone offset [26].

In the translator, we use three different helper functions to determine each offset (see listing 4.3).

```
let base_offset = function
  | Ast_src.C -> -9 | Ast_src.D -> -7 | Ast_src.E -> -5 | Ast_src.F ->
    -4
  | Ast_src.G -> -2 | Ast_src.A -> 0 | Ast_src.B -> 2

let acc_offset = function
  | Ast_src.Nat -> 0
  | Ast_src.Sharp -> 1
  | Ast_src.Flat -> -1

let oct_offset = function
  | Ast_src.Defined o -> (o - 4) * 12
  | _ -> 0
```

Listing 4.3: Matching the tone, accidental and octave to their respective offsets from the standard pitch

The sum of these offsets gives us the total offset O, which is used to adjust the standard pitch F_{std} to the specific tone. With this information, the output frequency F_{out} can be calculated using the following formula:

$F_{out} = F_{std} \cdot 2^{\frac{O}{12}}$

Then, the following formula is used to convert the frequency output F_{out} to a frequency F_n which is compatible with the SID chip [13, p. 187]:

$F_n = F_{out}/0.06097$

Finally, to divide this frequency into its high frequency F_{hi} and low frequency F_{lo}, the following calculations are made:

$F_{hi} = \lfloor F_n/256 \rfloor$
$F_{lo} = \lfloor F_n - (256 \cdot F_{hi}) \rfloor$

The only aspect of the note left to translate is its duration. In the source AST, a note only contains its fractional duration. Converting this to an absolute duration in frames[3] requires us to use both the tempo and time signature. To this end, a duration reference for the entire source file is calculated. This is simply the absolute duration of a 16th note. The function can be seen in figure 4.4.

```
    let get_duration_ref () =
        ...
        let bnv_duration = 3000 / tempo in
        match bnv with
        | 1 -> bnv_duration / 16
        | 2 -> bnv_duration / 8
        | 4 -> bnv_duration / 4
        | 8 -> bnv_duration / 2
        | 16 -> bnv_duration
```

Listing 4.4: Calculating the reference point for duration of notes

First, the amount of frames per minute (3000) is divided with the amount of beats per minute (tempo), resulting in the duration of a beat, or basic note[4]. Then, depending on the basic note value, this duration is divided to give the duration of a 16th note specifically.

Now, it is possible to multiply this duration to match it up with any note type, as seen in figure 4.5.

```
    let get_note_duration f =
        let duration_ref = get_duration_ref () in
        match f with
            | Ast_src.Whole -> duration_ref * 16
            | Ast_src.Half -> duration_ref * 8
            | Ast_src.Quarter -> duration_ref * 4
            | Ast_src.Eighth -> duration_ref * 2
            | Ast_src.Sixteenth -> duration_ref
```

Listing 4.5: Calculating the reference point for duration of notes

[3]20 millisecond intervals, as mentioned in section 3.3.6

[4]The basic note value is the second value of the time signature, and denotes the beats in which the tempo is counted (see Appendix B.4).

Returning to the example from Appendix H.2, we can use the first note of the first sequence, `c:1:4`, to demonstrate how it would be translated. Since its tone is 'c', the offset here would be -9 semitones. The lack of accidental and the fact that the octave is 4 means that there would be no additional offsets. Therefore, the resulting output frequency would be $F_{out} = 440 \cdot 2^{\frac{-9}{12}} \approx 261.63$. Converting to the SID-compatible frequency value gives us $F_n = 261.63/0.06097 \approx 4,291.05$. This results in the following high and low frequencies, which match the table in the Programmer's Reference Guide for the Commodore 64 [13, p. 385].

$F_{hi} = \lfloor 4,291.05/256 \rfloor = 16$
$F_{lo} = \lfloor 4,291.05 - (256 \cdot 16) \rfloor = 195$

As for the duration, we first use the tempo $t = 120$ and basic note value $b = 4$ to find the duration reference d:

$d = \lfloor \frac{3000}{\frac{t}{b}} \rfloor = \lfloor \frac{3000}{\frac{120}{4}} \rfloor = 6$

This means that every sixteenth note in this example will last 6 frames. Then, for the note `c:1:4` with fractional duration $f = 1$, the duration reference is simply multiplied by 4, giving us a duration of 24 frames.

This process continues for all notes in the source AST, as well as the previously mentioned more rudimentary translations, gradually constructing the new target AST. Its structure can be seen in figure 4.4.

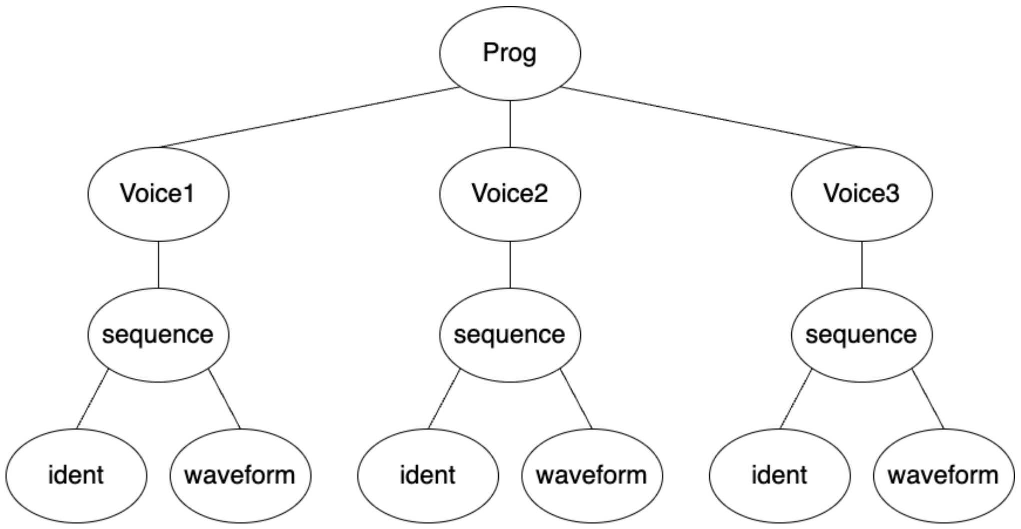


Figure 4.4: A visualization of the target AST made with our translator.

This AST is clearly more compact when compared to the previous iteration. This is in part due to parameters not being included. Since all notes have been translated to contain absolute frequencies and durations, the parameters have served their purpose and are no longer needed.

Even more noticeably, sequences are not depicted in this AST. Sequence identifiers are now merely tied to entries in the symbol table, wherein sequences have been updated to contain a list of notes in the updated format.

4.6 Code Generator

In the code generation phase, the target AST and the sequences stored in the symbol table are used to create the assembly file we compile to. The process starts by deleting the existing output file, if one exists, and then creating a new one. Afterwards, it inserts the music routine (see section 3.3.6) as a preamble[5] into the new output file. Following the preamble, it uses the target AST constructed by the translator and retrieves the symbol table containing sequences to write the assembly code, which defines the music data as defined in section 3.3.4. To ensure that the output file complies with 6502 assembly, an instruction set is defined, containing the entire 6502 instruction set, as well as assembler directives[6]. When constructing assembly instructions for the output file, this instruction set is used to ensure that an instruction's arguments comply with the specific instruction's maximum and minimum number of arguments. For general file operations, OCaml's standard library is used; this library allows for simple operations such as writing, appending, or deleting lines in a given file [28].

To generate the assembly code for the music data for the voices, the voices of the target AST are iterated through. For each iteration, using the sequence identifier and waveform, assembly instructions are created using the instruction set and written to the output file. Moreover, the function will write the voice's label, the instruction to repeat the voice, as well as the other instructions mandated by the music routine (see section 3.3.4), to the output file. As an example, looking at the code seen in H.2 with the following syntax:

```
    voice1 = [(seq1, vPulse)]
```

This is compiled into the following assembly code, with the correct voice label and sequence identifier. Moreover, in accordance with the music routines mandated structure, instructions to define the waveform, enter the sequence, and finally repeat the voice are created:

```
    voice1:         dc.b $FA
                    dc.b $FE
                    dc.w seq1
                    dc.b $FD
                    dc.w voice1
```

Similarly, another function iterates through the sequences stored in the symbol table to write the assembly code for the music data of the sequences. Each sequence contains a note list, which is iterated through, creating an assembly instruction for each note, with its high and low frequency values, as well as the duration. The values are converted to hexadecimal for the music routine to comprehend. For every instruction constructed, it is added to a list, which is finally iterated through and written to the output file. Finally, the instruction to end the sequence is created. Looking at the sequence definition for the code in the previous example, we see the following syntax:

```
    sequence seq1 = { c:1:4 }
```

[5]Introductory code, that is included at the beginning of a program

[6]Assembler directives are commands for the assembler that control its operation and settings, and do not get translated into executable machine instructions[27]

It is compiled into the following assembly code, with the correct sequence identifier and the hexadecimal values for high and low frequency and duration, calculated in the translator:

```
    seq1:           dc.b $C3, $10, $60
                    dc.b $FF
```

4.7 Assembler & Executable

After the code generation phase, an assembly output file is produced. This file cannot be directly executed on a Commodore 64 or an emulator, as it must first be converted into machine code. This conversion process is done through an assembler, and as mentioned in section 3.2, we have chosen to use *DASM*. The DASM assembler features, besides the 6502 instruction set, its own set of assembler directives, such as dc.b, but our compiler only utilises the following directives: `dc.b`, `dc.w`, `ORG`, and `PROCESSOR`. The first directive, `dc.b` stands for define constant bytes and defines a sequence of byte values, `dc.w` stands for define constant word and defines a sequence of word values, which are 16-bit. The last two directives are described in section 3.3.1. After DASM has finished processing the assembly file, it generates an executable file, which contains the machine code for the music piece.

The integration of the DASM assembler into the compiler is done through a system command call. This call is only triggered when the compiler is run with the `-DASM` flag. This is done to allow the end-user to choose if they want something that can be run directly.

The final phase of the compiler is the executable phase, where the compiled machine code can be executed directly on the hardware, or in our case, an emulator [4, p. 49]. We chose to execute our program on an emulator rather than an actual Commodore 64 computer for efficiency. Emulators are user-friendly as they can be kept locally, such that one can simply drag and drop the machine code of the compiled program into the emulator. This proves an optimised development phase as we have been able to test our compiler continuously and efficiently.

We selected *Vice*, a popular free cross-platform emulator, as our emulator of choice. Vice not only emulates the Commodore 64 but also the intricate features of its SID chip. Additionally, we opted for the reSID engine, a specific version of Vice that provides a much more accurate emulation of the SID chip. This version also allows us to select the exact SID model we want to emulate, which is the SID 6581. Another advanced feature of the reSID engine is the selection of different sampling methods for the SID signal output, such as fast, interpolation, and resampling. Of these, we have chosen to use the *resampling* setting for the execution of our program as it provides the best sound quality. Although this setting is rather CPU intensive, it is perfectly compatible with our program as it is non-interactive [29]. As opposed to DASM, Vice is not integrated into our compiler and must be run separately. Therefore, one has to transfer the machine code file generated from the compiler into the Vice emulator, which then generates the sound output.

4.8 Testing

Testing is an integral part of software development. By conducting tests, we aim to verify the functionality of our product by investigating whether functions or whole components work as expected. It is important to note that our tests will not prove the correctness of the program as it is difficult to test every single case, but by having broad test coverage we can instead provide an indication that the individual functions or components are functioning correctly, which gives us confidence that the overall program works as expected. This section will expand on how we have created unit, integration, and acceptance tests to attempt to ensure the quality of our product.

4.8.1 Unit Testing

Unit tests are tests made on smaller components of a larger program to establish that the components work correctly and as expected. Unit testing can uncover small errors in the logic of programs, whereby catching and correcting them can help to make a program more reliable, minimising unexpected errors [30]. For unit testing, we have used *Ounit2*, which is an OCaml testing framework that provides a test environment for grouping tests into suites and other functionalities like assert functions [31]. Our testing methodology has been to test functions after their creation, even though the best practice for testing is to do the opposite.

The general structure of our unit tests is to mock the data needed for the test, run the specific function, and finally use an assert function to verify that the result is as expected. An example of such a test can be seen in listing 4.6. This test aims to verify that the `note_translate` function in the AST translator correctly translates the source AST note to the target AST note. It creates a mock source AST note and a corresponding mock target AST note, which is the expected result of the function. Finally, the test asserts that the expected note is equal to the output of the translate function when applied to the source AST note.

```
let test_note_translate_sound _ctx =
    let note = AST_SRC.Sound (C, Nat, Whole, Defined 4) in
    let expected_note = {
        AST_TGT.highfreq = 16; AST_TGT.lowfreq = 195; AST_TGT.duration
    = 96
    } in
    assert_equal expected_note (AST_TRSL.note_translate note)
```

Listing 4.6: Unit test for the `note_translate` function

Instead of having to write the same code for mocking data in multiple tests, it is possible to use functions that set the testing environment by mocking any data to be used. This is how most of the symbol table tests have been conducted. For these tests, there is a setup and teardown function that retrieves the symbol table, clears it, and mocks a sequence. This sequence is passed to the test as parameters, which is run, and when the test is done, the symbol table is cleared again to ensure that each test is independent and not influenced by the previous test.

Finally, there are unit tests that aim to assert that the right exceptions are raised. An example of such a test can be seen in listing 4.7. The test uses the `assert_raises` function to assert that an exception with the correct error message is raised when a function is run. In the case of this test, the exception is expected to be raised as the same sequence is added to the symbol table twice.

```
let test_add_sequence2 id seq =
  SYM.add_sequence id seq;
  assert_raises (EXC.SyntaxErrorException "Sequences id's cannot be
   duplicated. Each sequence must have a unique id.")
     (fun () -> SYM.add_sequence id seq)

```

Listing 4.7: Unit test for the `add_sequence` function

4.8.2 Integration and Acceptance Testing

Having verified that components of the compiler behave as expected, their interaction will now be the focus. Integration tests aim to evaluate how components or modules of a larger system communicate and interact.

Our integration testing approach is to test the whole compiler as a single unit, and it involves compiling a series of programmes written in our source language, some expected to compile and some expected to raise specific exceptions. Inspecting the ASTs that can be displayed via a pretty printer function, the error messages, and the output file for these programmes enables verification that each component in the compilation process produces the expected result. The series of tests has been organised into folders to enable automation of the execution of the tests using GitHub Actions[7], which compiles all files within these folders.

One test defines a single sequence, with `seq1` as its identifier, containing one note. This test aims to verify that the compiler correctly parses the input file and produces a valid output file. Inspecting the source AST verifies that the sequence is properly parsed. Furthermore, inspecting the output file confirms that the sequence has been added, with the identifier `seq1`, indicating that the translator and the code generator functioned as expected. Finally, executing the generated executable on the Commodore 64 emulator confirms the playback of a single note.

Another failing test involves a negative octave value for a note. This test shows that our error handling could be improved. Compiling the file, the error message *"Lexical Error: Invalid input, expected a token at line 8 character 23"* is displayed. The lexer correctly identifies the error, but the error message could be more specific. For an overview of all integration tests, see Appendix G.1.

This testing approach is also used for our acceptance tests. Acceptance testing is a methodology that aims to verify that the entire product meets its specified requirements from a user's perspective. Typically, acceptance testing includes actual users, to enable the user to assert that the product works as intended during typical usage [33]. However, since our programming language is an intermediate

[7]GitHub Actions is a GitHub tool for automating and executing workflows directly in a repository [32]

language, user acceptance testing is not as relevant. As a result, acceptance tests are conducted using the programmes created by us, the developers, to simulate typical input. Therefore, the acceptance test approach has aimed to verify that the compiler produces the correct output for a specific program, adding confidence that the product will perform in production.

Ideally, a user acceptance test should have been performed on software developers with musical knowledge, to ensure that the source language's syntax is comprehensible and that its design provides a foundation for them to build their own musical tools. Moreover, the ideal integration testing process would be integrating components and performing tests step by step, as this would help pinpoint which specific components behave unexpectedly. However, as the results of the different components in the compilation process can be inspected, insight into the behaviour of smaller units is achievable. Additionally, the compiler's structured error handling enhances insight into component behaviour.

Chapter 5

Reflections

In this chapter, we will reflect on the final product by discussing the limitations of the compiler and then finally reflect on which features could be implemented in the future.

Sound Design

The ADSR and filter settings significantly influence the characteristic sound of chiptune music. Consequently, the limited scope of our syntax, which does not provide access to the full capabilities of the SID chip, presents major limitations. For instance, our compiler uses predefined ADSR settings, the filters are disabled, and the volume cannot be adjusted. Although you can produce interesting and complex melodies with our compiler, the product inherently limits the potential for varied and complex sounds compared to traditional methods. Therefore, adding syntax and functionality to adjust the filter and ADSR settings would be a major improvement to our compiler.

Compiler architecture

Limitations can also be found in the compiler's architecture. In its current state, two separate ASTs and a symbol table are used in the compilation process, with only minor changes from the first to the second AST. While it is standard practice to have multiple ASTs and a symbol table, our compiler could have used a single AST. Symbol tables are used to keep track of and store information through the compiler's different phases, but our compiler, which does not rely on variables, functions, or memory addresses, symbol table management becomes redundant. Therefore, a simpler solution would have been to store the sequences in the source AST, remove the target AST, and then perform the translations and transformations as a part of the code generation phase. This solution would have improved compilation time. However, future improvements to the language would add complexity to the compiler, meaning that the need for a second or multiple ASTs is likely. Therefore, while the compiler architecture might not be optimal, it provides a solid foundation for further development.

Testing with hardware

Having verified that the compiler's output, an executable file, can run on a Commodore 64 emulator, we then considered its execution on an actual Commodore 64. With this project's limited resources, getting hold of a Commodore 64 has not been possible, meaning that we cannot verify that the output can be executed on the actual hardware. If there ever was a possibility to test the compiler's output on the hardware, the method of testing it is not simple. It would require writing the executable file to a floppy disk, which then could be inserted into a floppy disk drive for the Commodore 64. Performing this test would certainly be interesting, and only after can we assert that our compiler is truly authentic to the Commodore 64 and its SID chip.

5.0.1 Error handling

Noget med error handling

5.1 Control structure

As previously mentioned, our current language is intermediate, the language serves as a foundation for other programmers who wish to create a language for the Commodore 64 without handling assembly and getting to know the SID chip. All though, in the future we could do a lot of these modifications ourselves, either by expanding our first AST or creating a brand new one.

Neptune does not have any control structures as programming languages often do, and this could be a major extension. Imagine writing a for-loop that repeats a certain node, or a section of nodes. Such control structures would be even more vital if Neptune had dynamic sound generation rather than static sound. With control structures and a interactive sound system we could make sounds based on actions made on the Commodore. Suppose a sound defined in a sequence is played if a user clicks the space button on the Commodore. Now imagine that the Commodore, while playing music, made graphics based on the sound it was playing. All this could be introduced with the implementation of control structures in Neptune by us, but as we have described Neptune as intermediate, it could also be introduced by others.

5.2 Percussion algorithm

The traditional method of introducing percussion to chiptune music, is to distribute the percussive elements to the three voices where there is space. Achieving this is possible with our source language, but the method is exhausting, as users must find or allocate the spaces themselves. Therefore, developing a feature where users could create a separate percussion sequence, which the compiler could distribute onto the three voices, would be an attractive improvement.

Appendices

Appendix A

The History of Chiptune

Appendix B

Music Theory

This Appendix expands on some of the theory of Western music used for developing Neptune.

B.1 Note Types

To compose the structure of a melody, its *rhythm*, a hierarchy of different notes is used (see figure B.1). A whole note can be divided into two half notes, each half note into two quarter notes, and so on. The fractional property of each note indicates its duration in accordance with the tempo (see section B.4). For each type of note, there also exists an equivalent rest symbol representing a pause [7, p. 9].

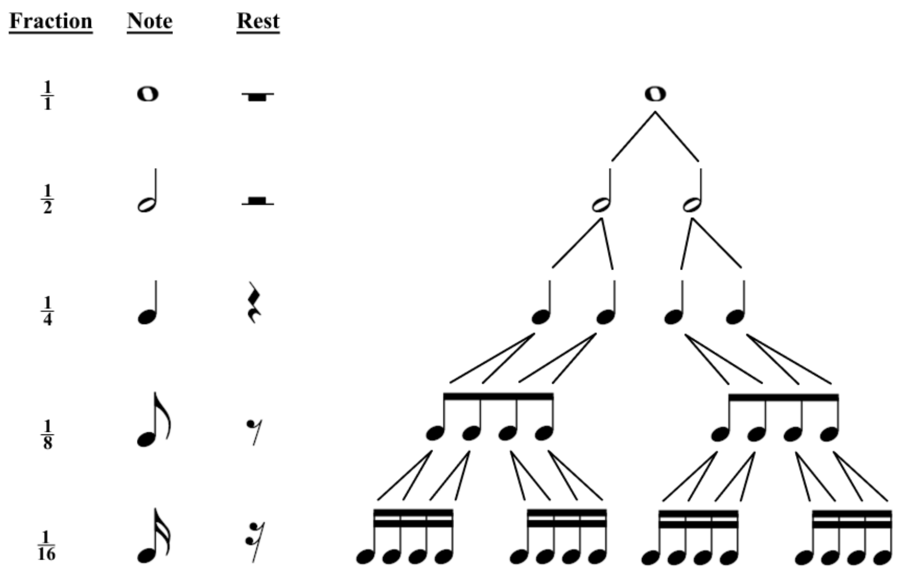


Figure B.1: An overview of the notes subdividing the whole note, their respective fractions, and equivalent rest symbols.

B.2 Time Signatures

To order a sequence of notes and rests over time, they are placed within a sequence of *measures*, which are separated by vertical bars. The measure delimitates the space available for notes with a *time signature*, expressed as you would a fraction. The upper value specifies the number of basic note values (beats) per measure, and the lower defines the basic note value itself [7, p. 11].

The most common time signature is 4/4, and thus it is often referred to as *common time* [34]. It has a basic note value of 4 (quarter note), with four of these quarter notes per measure (see figure B.2).

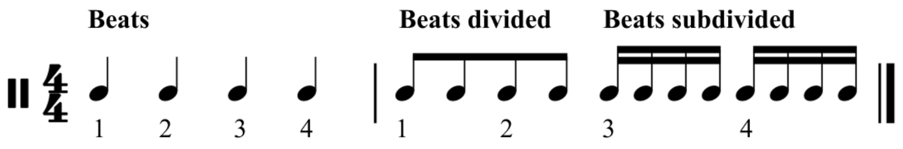


Figure B.2: Notation of two measures in common time

B.3 Tones

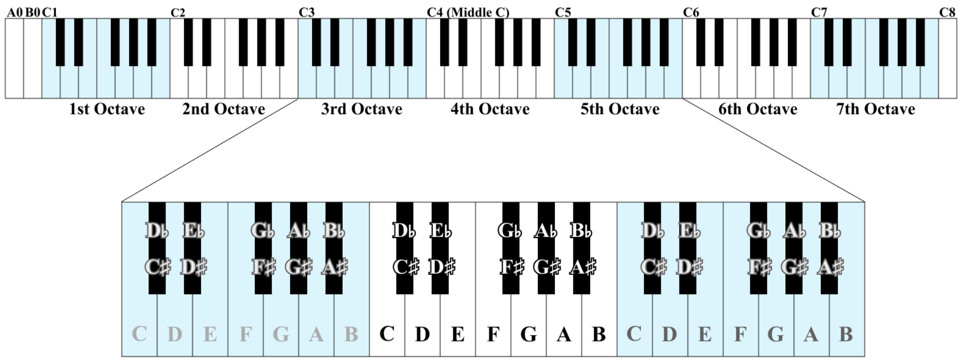


Figure B.3: This figure shows the structure of the twelve tones on a piano and how the structure repeats. The interval between two adjacent piano keys is called a semitone, also referred to as a half step [7, p. 7]

B.4 Tempo

The tempo of a piece of music assigns a duration of time to each of its beats. This is measured in the unit of beats per minute (BPM) [8]. For example, common time has a basic note value of 4, so the beats in this context are quarter notes. Given a tempo of 60 beats per minute, this means each quarter note will have a duration of one second. If the tempo were 120 BPM, each quarter note would last only half a second.

B.5 Pitch

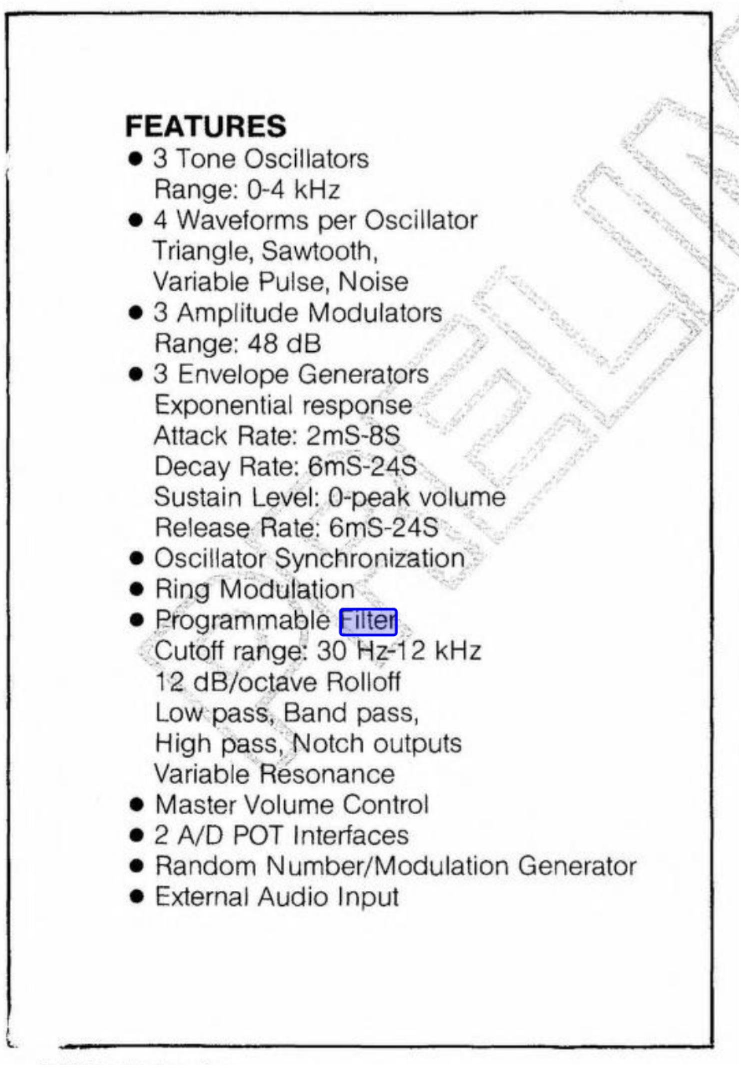
Each note represents a certain pitch, which is the highness or lowness of a tone. For example, playing the note A in the middle of a piano will produce a frequency of 440 Hz[1]. Playing a note an octave higher, its frequency is double that of the original note, so playing the A an octave above the middle A will produce a frequency of 880 hertz. The other notes in an octave are distributed with an equal ratio of frequency between them. This system of tuning is called the *equal temperament*.

[1]This frequency is often considered the standard pitch as it is the reference point for tuning musical instruments [9]

Appendix C

Hardware

C.1 SID 6581 Features

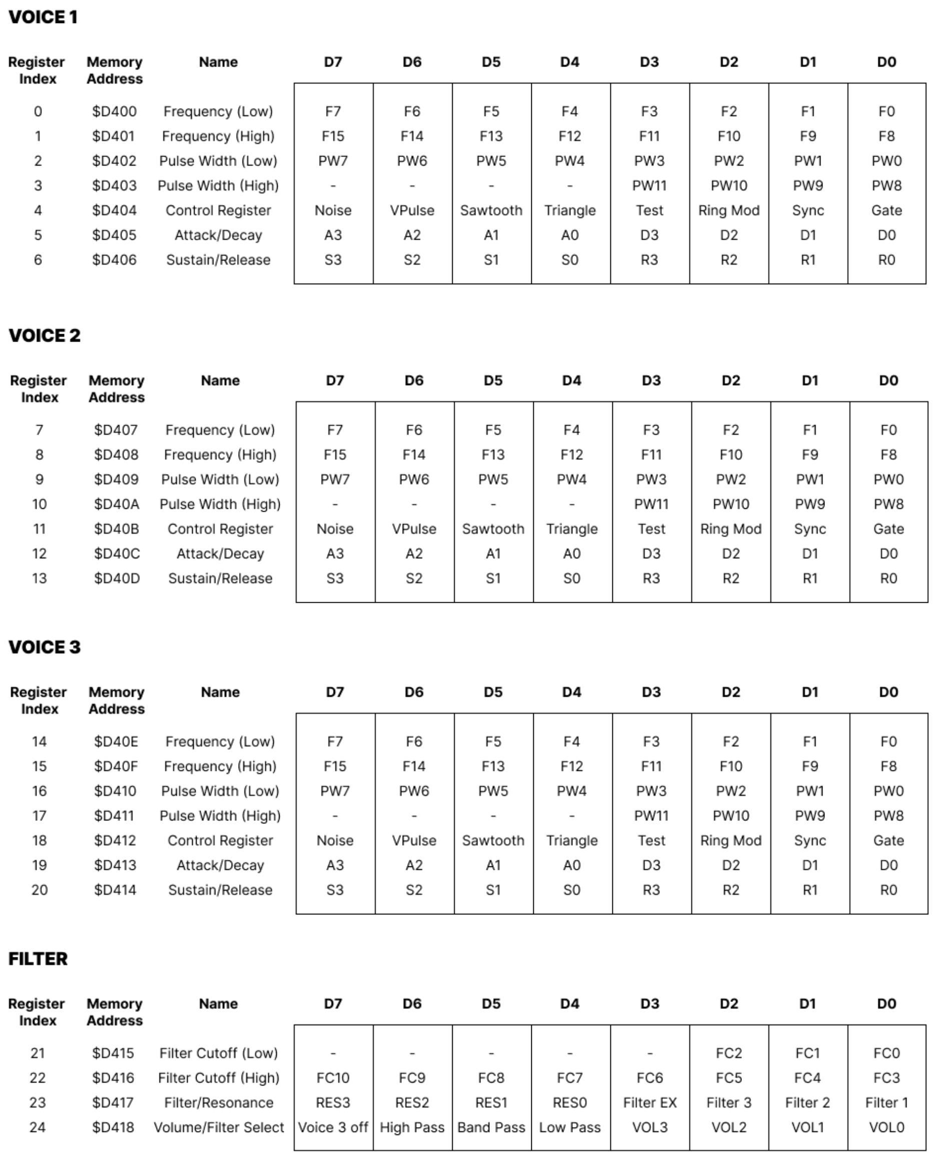


FEATURES

- 3 Tone Oscillators
  Range: 0-4 kHz
- 4 Waveforms per Oscillator
  Triangle, Sawtooth,
  Variable Pulse, Noise
- 3 Amplitude Modulators
  Range: 48 dB
- 3 Envelope Generators
  Exponential response
  Attack Rate: 2mS-8S
  Decay Rate: 6mS-24S
  Sustain Level: 0-peak volume
  Release Rate: 6mS-24S
- Oscillator Synchronization
- Ring Modulation
- Programmable Filter
  Cutoff range: 30 Hz-12 kHz
  12 dB/octave Rolloff
  Low pass, Band pass,
  High pass, Notch outputs
  Variable Resonance
- Master Volume Control
- 2 A/D POT Interfaces
- Random Number/Modulation Generator
- External Audio Input

Figure C.1: Complete list of the features of the SID 6581[11]

C.2 SID 6581 Control Registers



VOICE 1

Register Index	Memory Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
0	$D400	Frequency (Low)	F7	F6	F5	F4	F3	F2	F1	F0
1	$D401	Frequency (High)	F15	F14	F13	F12	F11	F10	F9	F8
2	$D402	Pulse Width (Low)	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
3	$D403	Pulse Width (High)	-	-	-	-	PW11	PW10	PW9	PW8
4	$D404	Control Register	Noise	VPulse	Sawtooth	Triangle	Test	Ring Mod	Sync	Gate
5	$D405	Attack/Decay	A3	A2	A1	A0	D3	D2	D1	D0
6	$D406	Sustain/Release	S3	S2	S1	S0	R3	R2	R1	R0

VOICE 2

Register Index	Memory Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
7	$D407	Frequency (Low)	F7	F6	F5	F4	F3	F2	F1	F0
8	$D408	Frequency (High)	F15	F14	F13	F12	F11	F10	F9	F8
9	$D409	Pulse Width (Low)	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
10	$D40A	Pulse Width (High)	-	-	-	-	PW11	PW10	PW9	PW8
11	$D40B	Control Register	Noise	VPulse	Sawtooth	Triangle	Test	Ring Mod	Sync	Gate
12	$D40C	Attack/Decay	A3	A2	A1	A0	D3	D2	D1	D0
13	$D40D	Sustain/Release	S3	S2	S1	S0	R3	R2	R1	R0

VOICE 3

Register Index	Memory Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
14	$D40E	Frequency (Low)	F7	F6	F5	F4	F3	F2	F1	F0
15	$D40F	Frequency (High)	F15	F14	F13	F12	F11	F10	F9	F8
16	$D410	Pulse Width (Low)	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
17	$D411	Pulse Width (High)	-	-	-	-	PW11	PW10	PW9	PW8
18	$D412	Control Register	Noise	VPulse	Sawtooth	Triangle	Test	Ring Mod	Sync	Gate
19	$D413	Attack/Decay	A3	A2	A1	A0	D3	D2	D1	D0
20	$D414	Sustain/Release	S3	S2	S1	S0	R3	R2	R1	R0

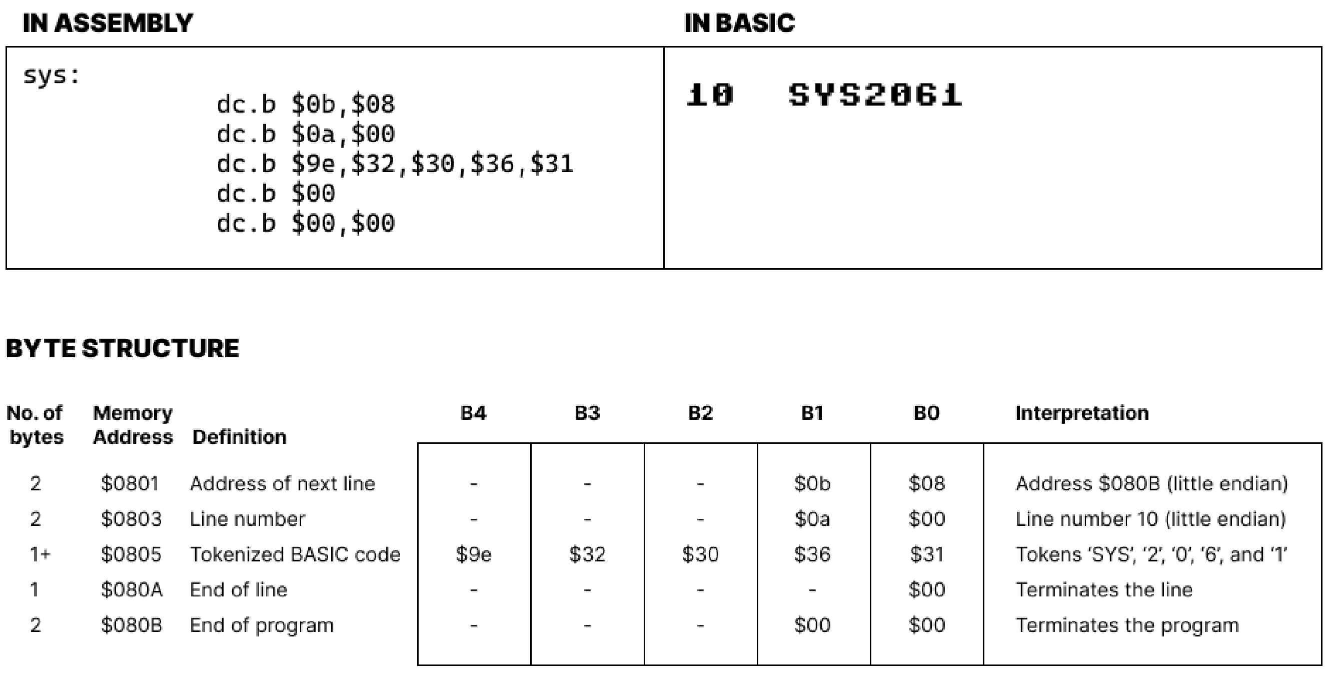
FILTER

Register Index	Memory Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
21	$D415	Filter Cutoff (Low)	-	-	-	-	-	FC2	FC1	FC0
22	$D416	Filter Cutoff (High)	FC10	FC9	FC8	FC7	FC6	FC5	FC4	FC3
23	$D417	Filter/Resonance	RES3	RES2	RES1	RES0	Filter EX	Filter 3	Filter 2	Filter 1
24	$D418	Volume/Filter Select	Voice 3 off	High Pass	Band Pass	Low Pass	VOL3	VOL2	VOL1	VOL0

Figure C.2: Overview of the different registers of the SID chip, showing its relative register index, absolute memory address in the C64, name of the register and the six data bits D7-D0 [11][35].

C.3 6502 Assembly & BASIC

C.4 Assembly Variables



IN ASSEMBLY

```
sys:
            dc.b $0b,$08
            dc.b $0a,$00
            dc.b $9e,$32,$30,$36,$31
            dc.b $00
            dc.b $00,$00
```

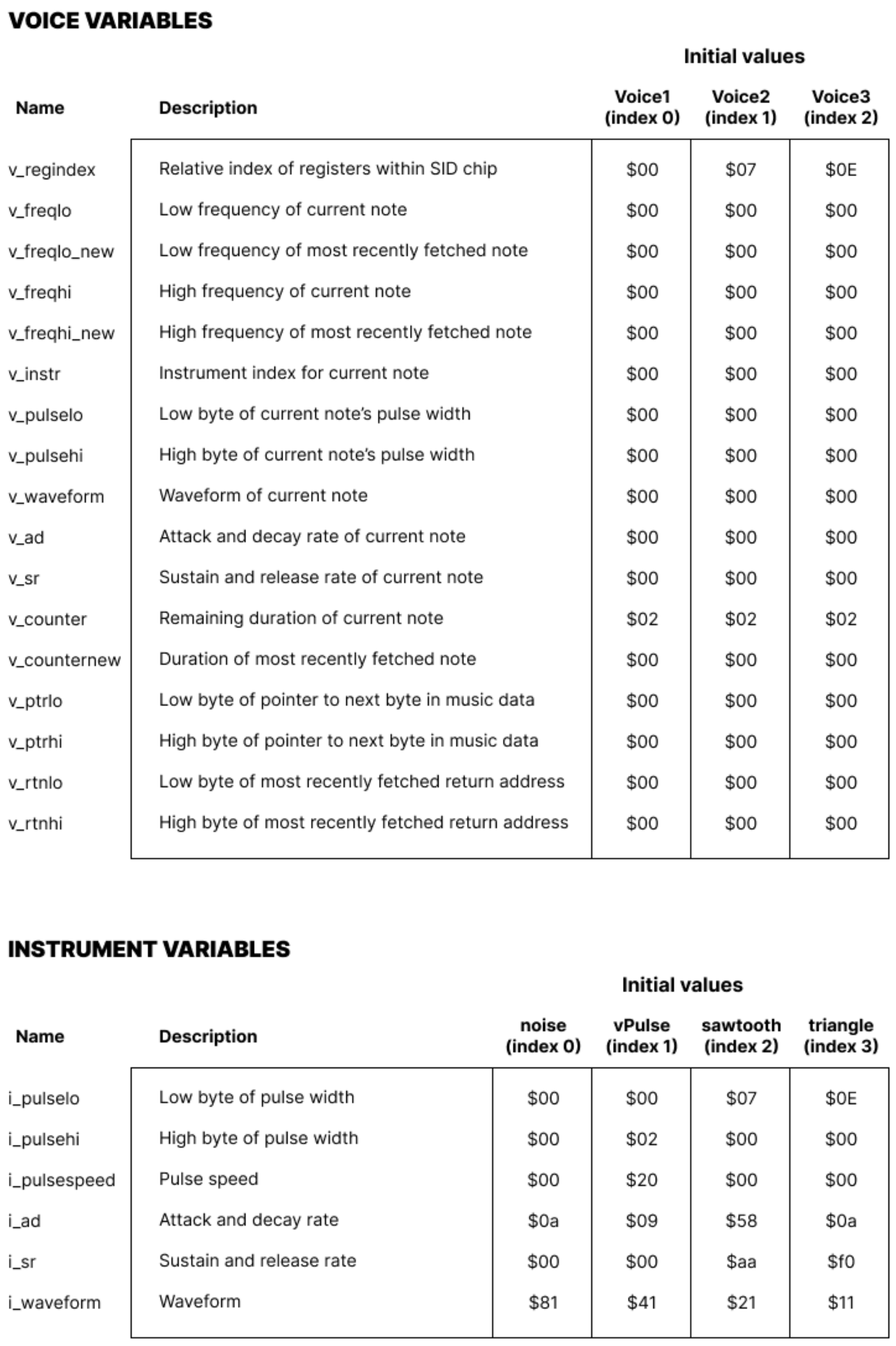
IN BASIC

```
10  SYS2061
```

BYTE STRUCTURE

No. of bytes	Memory Address	Definition	B4	B3	B2	B1	B0	Interpretation
2	$0801	Address of next line	-	-	-	$0b	$08	Address $080B (little endian)
2	$0803	Line number	-	-	-	$0a	$00	Line number 10 (little endian)
1+	$0805	Tokenized BASIC code	$9e	$32	$30	$36	$31	Tokens 'SYS', '2', '0', '6', and '1'
1	$080A	End of line	-	-	-	-	$00	Terminates the line
2	$080B	End of program	-	-	-	$00	$00	Terminates the program

Figure C.3: This figure shows how we use assembly 6502 byte data to structure a program in BASIC. This BASIC program is what executes in the C64 prompt and calls the remaining machine stored from memory at address $0801 (2061 in decimal, which is what is used in BASIC). First, a comparison is shown between the written assembly and the BASIC code it is interpreted as. Below, a description of the byte structure is elaborated on.



VOICE VARIABLES

Name	Description	Initial values Voice1 (index 0)	Voice2 (index 1)	Voice3 (index 2)
v_regindex	Relative index of registers within SID chip	$00	$07	$0E
v_freqlo	Low frequency of current note	$00	$00	$00
v_freqlo_new	Low frequency of most recently fetched note	$00	$00	$00
v_freqhi	High frequency of current note	$00	$00	$00
v_freqhi_new	High frequency of most recently fetched note	$00	$00	$00
v_instr	Instrument index for current note	$00	$00	$00
v_pulselo	Low byte of current note's pulse width	$00	$00	$00
v_pulsehi	High byte of current note's pulse width	$00	$00	$00
v_waveform	Waveform of current note	$00	$00	$00
v_ad	Attack and decay rate of current note	$00	$00	$00
v_sr	Sustain and release rate of current note	$00	$00	$00
v_counter	Remaining duration of current note	$02	$02	$02
v_counternew	Duration of most recently fetched note	$00	$00	$00
v_ptrlo	Low byte of pointer to next byte in music data	$00	$00	$00
v_ptrhi	High byte of pointer to next byte in music data	$00	$00	$00
v_rtnlo	Low byte of most recently fetched return address	$00	$00	$00
v_rtnhi	High byte of most recently fetched return address	$00	$00	$00

INSTRUMENT VARIABLES

Name	Description	Initial values noise (index 0)	vPulse (index 1)	sawtooth (index 2)	triangle (index 3)
i_pulselo	Low byte of pulse width	$00	$00	$07	$0E
i_pulsehi	High byte of pulse width	$00	$02	$00	$00
i_pulsespeed	Pulse speed	$00	$20	$00	$00
i_ad	Attack and decay rate	$0a	$09	$58	$0a
i_sr	Sustain and release rate	$00	$00	$aa	$f0
i_waveform	Waveform	$81	$41	$21	$11

Figure C.4: These tables depict all variables within our assembly code, as well as the initial values they contain.

Appendix D

Sound Waves

Sound is caused by vibrations, and these vibrations are transmitted through the air in the form of sound waves. The sound waves are then simply a displacement of molecules in the medium from their resting position and back [36]. In figure D.1 such a graphical representation of the displacement and change over time is shown, with the x-axis typically representing time and the y-axis representing the amplitude of the displacement.

D.1 Amplitude

can be seen as the maximum displacement of a given sound wave and is measured from zero to the maximum amount displaced (see figure D.1). To simplify, the amplitude can be thought of as loudness, with an amplitude of 0, a sound wave following the X-axis would represent complete silence. In contrast a higher amplitude would produce an actual sound and would increase in loudness if amplitude was increased[37].

D.2 Frequency

Frequency measures how many times a waveform iterates in a certain period of time, and is often measured in Hz (Hertz), the number of repetitions per second. The waveform seen in figure D.1 is periodic, meaning that it repeats at regular intervals over time indefinitely. These are typically generated with a computer, as instruments and voices tend to be more complex, producing waveforms that contain multiple frequencies [37].

D.3 Harmonics

Harmonics are additional waveforms with different frequencies that are created by certain waveforms. This is because complex waveforms can be described as a bunch of different sine waves layered on top of each other, making the sine wave the fundamental waveform[1] (see Section **??**) [37]. It is possible to hear the constant frequency of a sine wave, while other waveforms have varying harmonic

[1] also known as the root frequency

frequencies that contribute to their overall sound. Harmonics are always multiples of the fundamental frequency. In even harmonics, if the fundamental frequency is 1 Hz, the second harmonic is 2 Hz, the fourth harmonic 4 Hz, and this pattern continues to infinity. Odd harmonics includes the root frequency, the 3rd, 5th harmonics, and so on.

D.4 Waveforms

The sine wave (see figure D.1), as stated earlier, is the fundamental waveform, it is a smooth periodic wave that moves between positive and negative values. It is defined by the sine function $y(t) = A \sin 2\pi f t + \phi$ in which A is the amplitude, f is the frequency and ϕ is the phase[2]. Sine waves are important for synthesising audio, because they, as stated in Fourier's theorem(see section D.5), can recreate any other periodic waveform [38]. They produce a clear tone without any harmonics.

D.5 Sine Waves

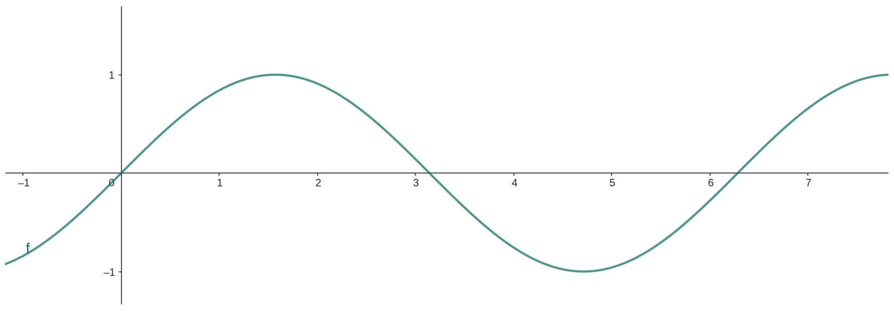


Figure D.1: Simple sine wave, described by $f(x) = sin(x)$

Note

Fourier's theorem states "Any periodic signal is composed of a superposition of pure sine waves, with suitably chosen amplitudes and phases, whose frequencies are harmonics of the fundamental frequency of the signal." [39] Following this theorem, a periodic square wave or any other periodic waves can be described as a series of sinusoidal functions.

In the following sections, the waveforms are described using additive synthesis [40]. This method of synthesising waveforms is **not** the method used inside of the SID 6581 chip [11], which is subtractive synthesis. Although due to subtractive synthesis being more abstract when needing to visualise it, due to filtering frequencies makes it harder to visualise, we have chosen to use additive synthesis to illustrate the waveforms in our project.

D.5.1 The Variable Pulse Waveform

The variable pulse waveform (see figure **??**) is a periodic waveform that alternates between two levels of amplitude, high and low. The shape of the variable pulse waveform is determined by the *duty*

[2]Phase is the timing of when the wave cycle begins.

cycle, which is the ratio of the time before switching between the two levels. In relation to harmonics, the variable pulse harmonics are dependent on the duty cycle and is not fixed like with triangle or sawtooth. The sound produced by the waveform is also duty cycle dependent, A higher duty cycle produces a hollow sound, while a smaller one turns it into a more thinner sound [41].

D.5.2 The Triangle Waveform

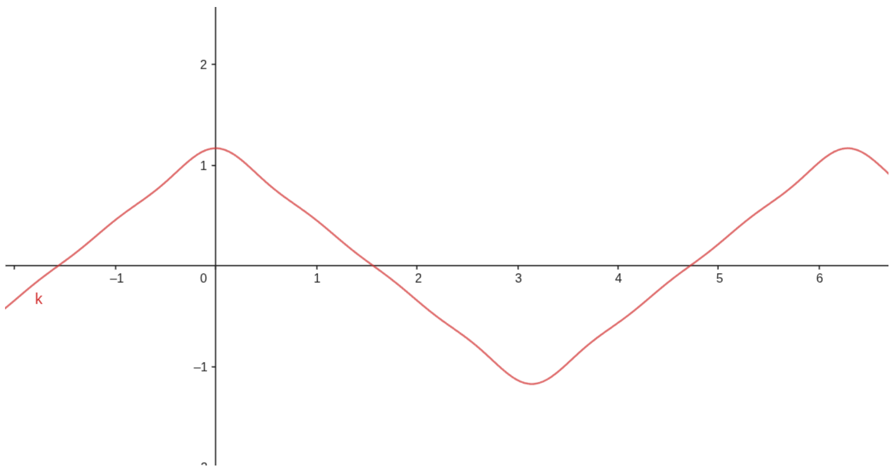


Figure D.2: Simple Triangle wave, additive synthesis[41]

The triangle waveform (see figure D.2) is periodic and has a triangle shape. An ideal triangle wave mostly consists of odd harmonics and the amplitude of these harmonics decreases to the square of the harmonic number, thus making the 3rd harmonic 1/9th of the amplitude of the fundamental. This waveform has a flute-like and pure sounding tone [41].

D.5.3 The Sawtooth Waveform

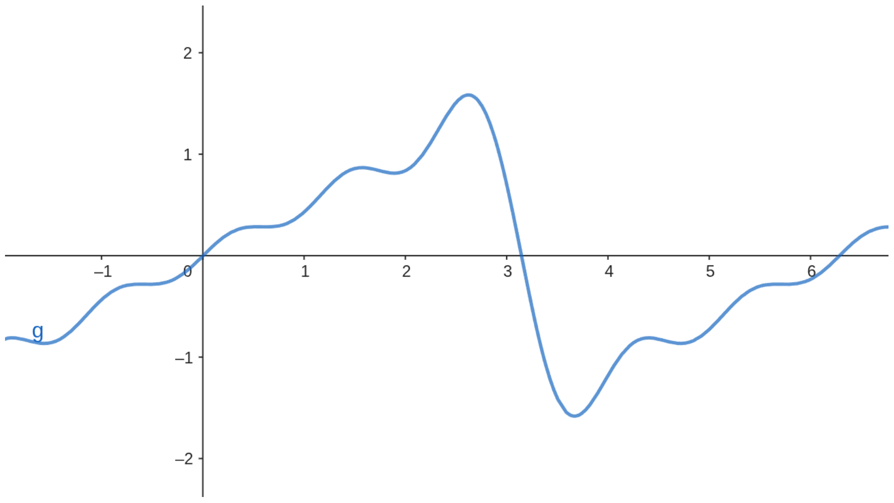


Figure D.3: Simple sawtooth wave, additive synthesis with 4 harmonics

The sawtooth waveform (see figure D.3) is periodic and has a ramp shape that rises. It contains both odd and even harmonics and their amplitudes decrease inversely in relation to the harmonic numbers.

The sawtooth wave has many harmonic frequencies, making it bright and buzzy [42].

D.5.4 The Noise Waveform

The noise waveform (see figure **??**) is a non-periodic waveform which has a more random pattern or has a completely pseudo-random pattern over time. In terms of usage in sound synthesis, it has a more instrumental sound which can be for more percussive sounds such as snares or hi-hats [41].

Appendix E

MoSCoW Specifications

We use the MoSCoW model to prioritise and identify the specifications of our compiler. The model is divided into four sub-categories, each representing the level of prioritisation, as follows: *Must Haves, Should Haves, Could Haves,* and *Will Not Haves* [43].

Must Haves

ID	**Description**	**Traceability**
M01	The compiler must take a Neptune source file as input, and output an assembly file	Section 4.2
M02	The compiler must have a lexer that performs a lexical analysis on the input file/source language and outputs a stream of tokens	Section 4.2
M03	The compiler must translate the high-level AST generated from the source language into a low-level AST resembling the target language	Section 4.2
M04	The assembly code output by the compiler must be assembled into machine code by an external assembler	Section 4.2
M05	The compiled program must be executable on an external Commodore 64 emulator and generate proper sound output	Section 4.2
M06	Precise and informative compilation errors must be raised if compilation fails	Section **??**

Figure E.1: The must have specifications are the most essential requirements for creating a minimum viable product. These requirements define the functionalities which the product should guarantee to deliver.

Should Haves

ID	Description	Traceability
S01	The compiled program should utilise all three of the SID's voices	Section 3.1
S02	The compiled program should support multiple voices playing simultaneously, providing a complex sound output coherent with the SID 6581	Section 3.1.1
S03	The compiled program should support various distinct sequences to be played simultaneously by different voices	Section **??**
S04	The compiled program should be able to generate sound output of varying waveforms within each voice	Section 3.1.1
S05	An assembler should be integrated in the compiler	Section 4.2

Figure E.2: The should have specifications are non-essential features, meaning they are not necessary for the basic functionalities of the product, but add significant features and optimisation.

Could Haves

ID	Description	Traceability
C01	The compiled program could accommodate user-defined envelopes	Section 3.1.3
C02	The compiled program could change ADSR values dynamically	Section 3.1.3
C03	The compiled program could handle user-defined frequency cutoff filters	Section 3.1.4

Figure E.3: The could have specifications are features that would enhance the product if implemented, but would not have a significant impact if left out.

Will Not Haves

ID	Description	Traceability
W01	The compiled program will not accommodate user interaction when run on the Commodore 64 emulator, as the focus of this project is utilisation of the hardware's sound generation capabilities	N/A
W02	The compiled program will not utilise the 'fourth' voice, allowing for more than three voices, as we follow the limitations of the SID 6581. As the fourth voice is not an actual voice but rather a glitch, we have decided not to include a fourth voice in our program	N/A

Figure E.4: The will not have specifications are features that we have decided not to prioritise for this project.

Appendix F

Lexer & Parser

F.1 Regular Expressions & Lexing Rules

```
let digit = ['0'-'9']
let int = digit+

let whitespace = [' ' '\t' '|']+
let newline = "\r\n" | '\n' | '\r'
let tone = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "r"
let letter = ['a'-'z' 'A'-'Z']
let ident = letter (letter | '-' | digit)+
```

Listing F.1: Regular Expressions

```
rule read = parse
    | whitespace {read lexbuf}
    | newline {Lexing.new_line lexbuf; read lexbuf}
    | tone {TONE (Lexing.lexeme lexbuf)}
    | ident as s {ident_or_keyword s}
    | int {INT (int_of_string (Lexing.lexeme lexbuf))}
    | "/*" {comment lexbuf}
    | "#"  {SHARP}
    | "_" {FLAT}
    | "{" {LCB}
    | "}" {RCB}
    | "[" {LSB}
    | "]" {RSB}
    | "(" {SP}
    | ")" {EP}
    | ":" {COLON}
    | "," {COMMA}
    | "=" {ASSIGN}
    | eof {EOF}
    | _ {raise (LexicalErrorException
        ("Invalid input, expected a token " ^ lexeme_error lexbuf))}

(...)

and comment = parse
    | "*/" {read lexbuf}
    | newline {unterminated_comment lexbuf}
    | _ {comment lexbuf}
    | eof {unterminated_comment lexbuf}
```

Listing F.2: Lexing Rules

F.2 Parse Tree

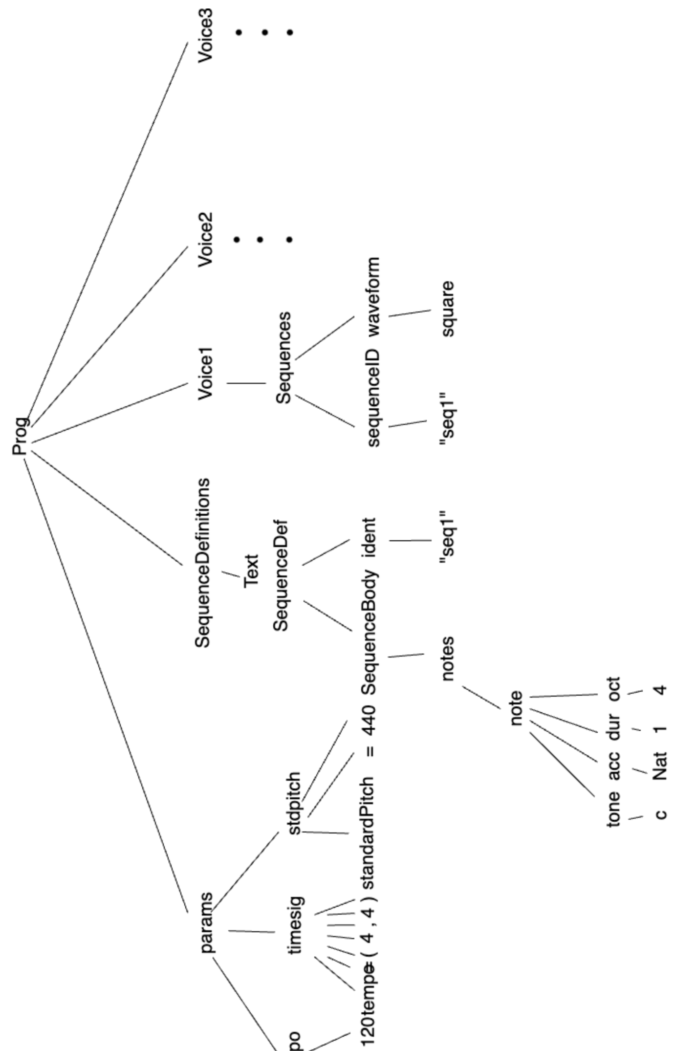


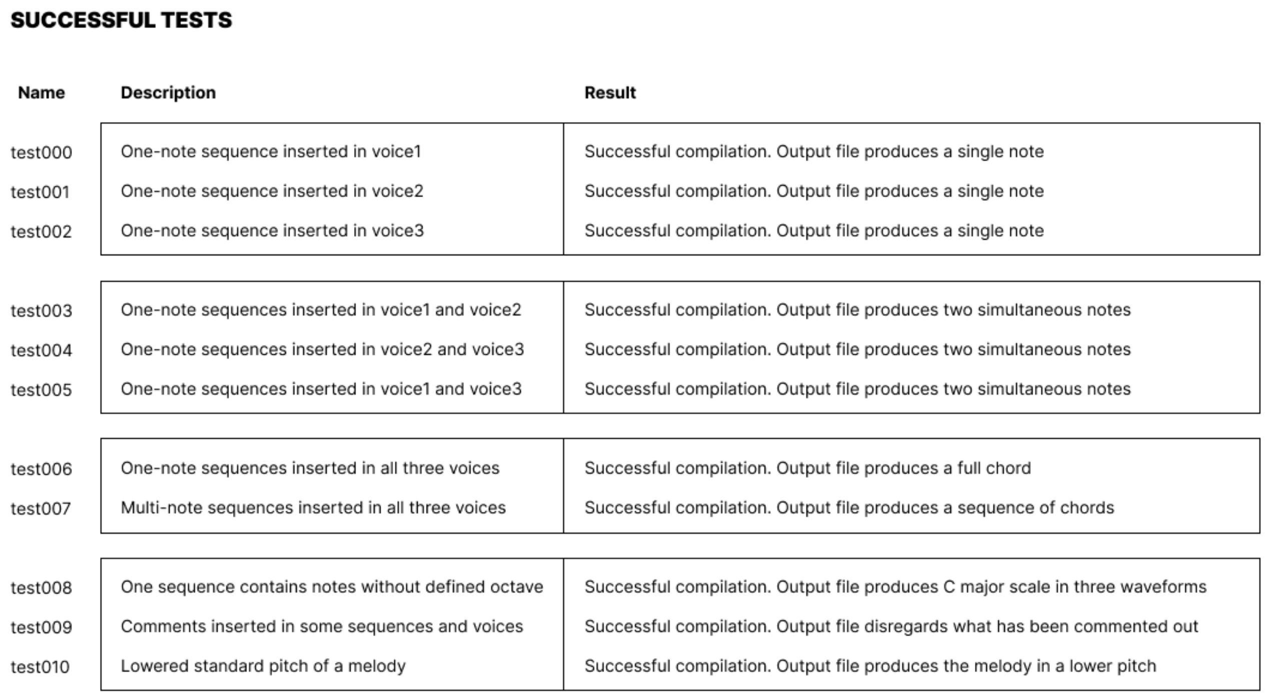
Figure F.1: A simplified parse tree representing the source code example in appendix **??**. Voice2 and and Voice3 are very similar to Voice1, and therefore not specified

Appendix G

Testing

G.1 Integration and Acceptance Tests

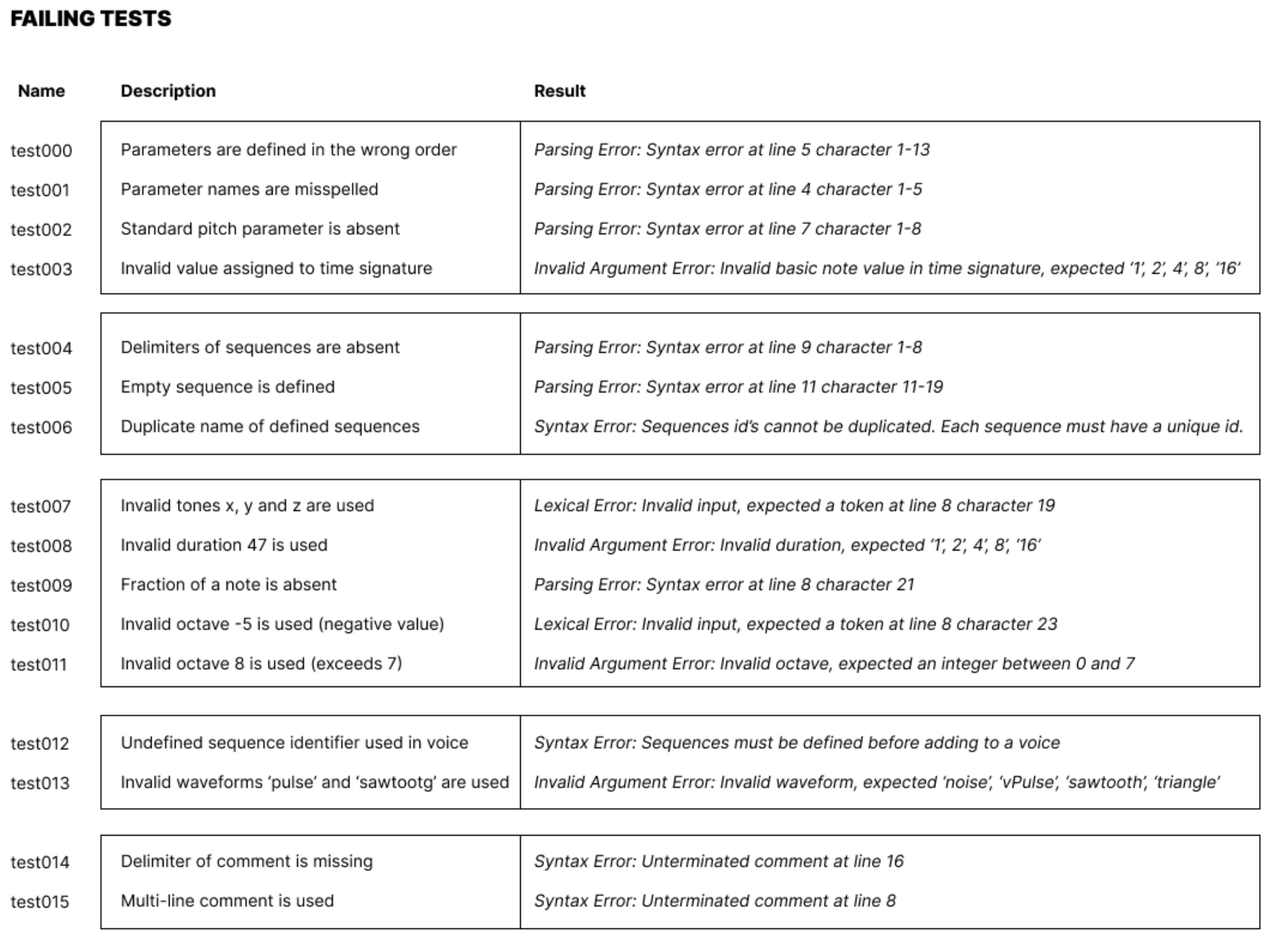
This section presents the results of the integration and acceptance tests. They are organised into successful tests that are expected to compile in Figure G.1 and failing tests that are expected to display an error message in Figure G.2. The figures show all tests' names, descriptions, and results.



SUCCESSFUL TESTS

Name	Description	Result
test000	One-note sequence inserted in voice1	Successful compilation. Output file produces a single note
test001	One-note sequence inserted in voice2	Successful compilation. Output file produces a single note
test002	One-note sequence inserted in voice3	Successful compilation. Output file produces a single note
test003	One-note sequences inserted in voice1 and voice2	Successful compilation. Output file produces two simultaneous notes
test004	One-note sequences inserted in voice2 and voice3	Successful compilation. Output file produces two simultaneous notes
test005	One-note sequences inserted in voice1 and voice3	Successful compilation. Output file produces two simultaneous notes
test006	One-note sequences inserted in all three voices	Successful compilation. Output file produces a full chord
test007	Multi-note sequences inserted in all three voices	Successful compilation. Output file produces a sequence of chords
test008	One sequence contains notes without defined octave	Successful compilation. Output file produces C major scale in three waveforms
test009	Comments inserted in some sequences and voices	Successful compilation. Output file disregards what has been commented out
test010	Lowered standard pitch of a melody	Successful compilation. Output file produces the melody in a lower pitch

Figure G.1: An overview of the tests that are expected to succeed



FAILING TESTS

Name	Description	Result
test000	Parameters are defined in the wrong order	*Parsing Error: Syntax error at line 5 character 1-13*
test001	Parameter names are misspelled	*Parsing Error: Syntax error at line 4 character 1-5*
test002	Standard pitch parameter is absent	*Parsing Error: Syntax error at line 7 character 1-8*
test003	Invalid value assigned to time signature	*Invalid Argument Error: Invalid basic note value in time signature, expected ‘1’, 2’, 4’, 8’, ‘16’*
test004	Delimiters of sequences are absent	*Parsing Error: Syntax error at line 9 character 1-8*
test005	Empty sequence is defined	*Parsing Error: Syntax error at line 11 character 11-19*
test006	Duplicate name of defined sequences	*Syntax Error: Sequences id’s cannot be duplicated. Each sequence must have a unique id.*
test007	Invalid tones x, y and z are used	*Lexical Error: Invalid input, expected a token at line 8 character 19*
test008	Invalid duration 47 is used	*Invalid Argument Error: Invalid duration, expected ‘1’, 2’, 4’, 8’, ‘16’*
test009	Fraction of a note is absent	*Parsing Error: Syntax error at line 8 character 21*
test010	Invalid octave -5 is used (negative value)	*Lexical Error: Invalid input, expected a token at line 8 character 23*
test011	Invalid octave 8 is used (exceeds 7)	*Invalid Argument Error: Invalid octave, expected an integer between 0 and 7*
test012	Undefined sequence identifier used in voice	*Syntax Error: Sequences must be defined before adding to a voice*
test013	Invalid waveforms ‘pulse’ and ‘sawtootg’ are used	*Invalid Argument Error: Invalid waveform, expected ‘noise’, ‘vPulse’, ‘sawtooth’, ‘triangle’*
test014	Delimiter of comment is missing	*Syntax Error: Unterminated comment at line 16*
test015	Multi-line comment is used	*Syntax Error: Unterminated comment at line 8*

Figure G.2: An overview of the tests that are expected to fail

Appendix H

Source Code

H.1 Tetris Source Code

```
tempo = 150
timeSignature = (2,4)
standardPitch = 440

sequence main = {
    e:4:  b:8:3 c:8: | d:8: e:16: d:16: c:8: b:8:3 |
    a:4:3 a:8:3 c:8: | e:4:  d:8  c:8  |
    b:4:3 r:8   c:8  | d:4             e:4         |
    c:4   a:4:3      | a:4:3 r:4       |
    r:8 d:4:    f:8: | a:4:            g:8: f:8:   |
    e:4:  r:8   c:8: | e:4:  d:8: c:8: |
    b:4:3 r:8   c:8: | d:4:            e:4:        |
    c:4:  a:4:3      | a:4:3 r:4       |
}

sequence bridge = {
    e:2: | c:2: | d:2: | b:2:3 |
}

sequence bridgeA = {
    c:2: | a:2:3 | g#:4:3 b:4:3 | e:4: r:4|
}

sequence bridgeB = {
    a:4:3 e:4: | a:2: | g#:2 | r:2 |
}

sequence bass1 = {
    e:8:2  e:8:3  e:8:2  e:8:3  | e:8:2  e:8:3  e:8:2  e:8:3  |
    a:8:1 a:8:2 a:8:1 a:8:2     | a:8:1 a:8:2 a:8:1 a:8:2     |
    g#:8:1 g#:8:2 g#:8:1 g#:8:2 | g#:8:1 g#:8:2 g#:8:1 g#:8:2 |
    a:8:1 a:8:2 a:8:1 a:8:2     | a:8:1 a:8:1 b:8:1 c:8:2     |
}

sequence bass2 = {
    d:8:2 d:8:3 d:8:2 d:8:3 | d:8:2 d:8:3 d:8:2 d:8:3 |
    c:8:2 c:8:3 c:8:2 c:8:3 | c:8:2 c:8:3 c:8:2 c:8:3 |
    b:8:1 b:8:2 b:8:1 b:8:2 | b:8:1 b:8:2 b:8:1 b:8:2 |
    a:8:1 a:8:2 a:8:1 a:8:2 | a:8:1 a:8:2 a:8:1 a:8:2 |
}

sequence bass3 = {
    a:8:1 a:8:2 a:8:1 a:8:2     | a:8:1 a:8:2 a:8:1 a:8:2     |
    g#:8:1 g#:8:2 g#:8:1 g#:8:2 | g#:8:1 g#:8:2 g#:8:1 g#:8:2 |
    a:8:1 a:8:2 a:8:1 a:8:2     | a:8:1 a:8:2 a:8:1 a:8:2     |
    g#:8:1 g#:8:2 g#:8:1 g#:8:2 | r:8    g#:8:2 g#:4:2        |
}

sequence kickHit = { d:16:0 }
sequence kickResonate = { d:8:0 r:16 }
sequence snare = { a:8:5 r:8 }

voice1 = [(main, triangle), (bridge, triangle), (bridgeA, triangle),
          (bridge, triangle), (bridgeB, triangle)]

voice2 = [(bass1,sawtooth), (bass2, sawtooth),
          (bass3, sawtooth), (bass3, sawtooth)]

voice3 = [(kickHit,noise), (kickResonate,vPulse), (snare, noise)]
```

Listing H.1: Tetris Source code example for Neptune

H.2 Mini Source Code

```
tempo = 120
timeSignature = (4,4)
standardPitch = 440

sequence seq1 = { c:1:4 }
sequence seq2 = { d:4:4 e_:4:4 }
sequence seq3 = { g:8:3 f#:8:3 a:4:3 }


```

```
voice1 = [(seq1, square)]
voice2 = [(seq2, triangle), (seq2, triangle)]
voice3 = [(seq3, sawtooth), (seq2, noise)]
```

Listing H.2: Small source code example for Neptune

Bibliography

Bibliography

[1] Kenneth B. McAlpine, *Bits and Pieces: A History of Chiptunes*. New York, NY : Oxford University Press, 2018, [Accessed 2025 Feb 23].

[2] D. Schwebel, “A Guide to Chiptune Music,” https://www.synthcentric.com/articles/a-guide-to-chiptune-music, [Accessed 2025 Feb 22].

[3] Vinicius Fulber-Garcia, “Imperative and Declarative Programming Paradigms,” https://www.baeldung.com/cs/imperative-vs-declarative-programming, 2024, [Accessed 2025 May 7].

[4] R. W. Sebesta, *Concepts of Programming languages*, 11th ed. Pearson Education Limited, 2016, [Accessed 2025 Apr 11].

[5] “LilyPond - Music notation for everyone,” https://lilypond.org/index.html, [Accessed 2025 May 8].

[6] C. Wright, “Lecture 5 - Melody: Notes, Scales, Nuts and Bolts,” https://oyc.yale.edu/music/musi-112/lecture-5#, 2012, [Accessed 2025 Feb 28].

[7] B. Benward and M. Saker, *Music in Theory and Practice*, M. Ryan, Ed. McGraw-Hill, 2008.

[8] S. Chase, “What Is Tempo In Music? A Complete Guide,” https://hellomusictheory.com/learn/tempo/, 2024, [Accessed 2025 Feb 23].

[9] A. Steen, “432 vs. 440 Hz Frequencies: A Comprehensive Examination,” https://primesound.org/432-vs-440, 2024, [Accessed 2025 Feb 21].

[10] H. Hüttel, *Transition and Trees: An Introduction to Structural Operational Semantics*. Cambridge, 2010, [Accessed 2025 May 7].

[11] *6581 Sound Interface Device (SID)*, https://archive.org/details/mos_6581_sid_preliminary_october_1982/mode/2up, Commodore Semiconductor Group, 1982, [Accessed 2025 Mar 10].

[12] M. DeVoto, “polyphony,” https://www.britannica.com/art/polyphony-music, 2023, [Accessed 2025 Feb 27].

[13] *Commodore 64 Programmer’s Reference Guide*, https://archive.org/details/c64-programmer-ref/, Commodore Computer, 1982, [Accessed 2025 Mar 12].

[14] A. Price, “How to use basic ADSR filter envelope parameters,” https://www.musicradar.com/tuition/tech/how-to-use-basic-adsr-filter-envelope-parameters-578874, 2023, [Accessed 2025 Mar 12].

[15] J. Butterfield, *Machine Language for the Commodore 64, 128, and Other Commodore Computers*. Prentice Hall Press, 1986.

[16] “Remembering Computing Legend Thomas Kurtz,” https://fas.dartmouth.edu/news/2024/11/remembering-computing-legend-thomas-kurtz, 2025, [Accessed 2025 May 12.

[17] Randal E. Bryant and David R. O’Hallaron, *Computer Systems: A Programmer’s Perspective 3rd Edition*. Pearson Education Limited, 2016, [Accessed 2025 Mar 06].

[18] “6502 Architecture,” http://www.6502.org/users/obelisk/6502/architecture.html, 2002, [Accessed 2025 May 20].

[19] “First 6502 assembly program,” http://forum.6502.org/viewtopic.php?t=4948, 2017, [Accessed 2025 May 20].

[20] Léon Gondelman, “Languages and Compilation - Lecture 1 - Introduction,” https://homes.cs.aau.dk/~lego/compil25/lectures/1/lecture1-pp.pdf, 2025, [Accessed 2025 May 16].

[21] Léon Gondelman, “Languages and Compilation - Lecture 2 - Abstract Syntax, Semantics,” https://homes.cs.aau.dk/~lego/compil25/lectures/2/lecture2-pp.pdf, 2025, [Accessed 2025 May 16].

[22] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, “The OCaml Manual,” https://ocaml.org/manual/5.3/index.html#, 2025, [Accessed 2025 May 12].

[23] C. N. Fischer, R. K. Cytron, and J. Richar J. Leblanc, *Crafting a Compiler*. Pearson Education Limited, 2010, [Accessed 2025 May 5].

[24] L. Gondelmann, “Languages and Compilation - Lecture 4 - Lecture 4 - Parsing, Part 1,” https://homes.cs.aau.dk/~lego/compil25/index.html, 2025, [Accessed 2025 May 12].

[25] F. Pottier, “Menhir: A LR(1) parser generator,” https://gallium.inria.fr/~fpottier/menhir/, 2024, [Accessed 2025 May 12].

[26] M. Délèze, “Calculation of the frequency of the notes of the equal tempered scale,” https://www.deleze.name/marcel/en/physique/musique/Frequences-en.pdf, [Accessed 2025 Feb 21].

[27] “2.3: Assembler Directives,” https://eng.libretexts.org/Bookshelves/Electrical_Engineering/Electronics/Implementing_a_One_Address_CPU_in_Logisim_(Kann)/02%3A_Assembly_Language/2.03%3A_Assembler_Directives, 2023, [Accessed 2025 May 15].

[28] “File Manipulation,” https://ocaml.org/docs/file-manipulation#, [Accessed 2025 May 13].

[29] “The Versatile Commodore Emulator,” https://vice-emu.sourceforge.io/vice_toc.html, 2024, [Accessed 2025 May 19].

[30] K. Rana, “Unit Testing,” https://artoftesting.com/unit-testing#Best_Practices_for_Unit_Testing, 2025, [Accessed 2025 May 9.

[31] M.-M. Zeeman and S. L. Gall, “OUnit: xUnit testing framework for OCaml,” https://ocaml.org/p/ounit2/2.2.3/doc/index.html#ounit:-xunit-testing-framework-for-ocaml, [Accessed 2025 May 9.

[32] “GitHub Actions documentation,” https://docs.github.com/en/actions, [Accessed 2025 May 14].

[33] “Acceptance testing,” https://en.wikipedia.org/wiki/Acceptance_testing, 2025, [Accessed 2025 May 11.

[34] M. Aichele, “Understanding Time Signatures and Meters: A Musical Guide,” https://www.libertyparkmusic.com/musical-time-signatures/, 2018, [Accessed 2025 Feb 23].

[35] S. Leemon, *Mapping the Commodore 64 and 64C*, 1st ed. Compute!, 1987.

[36] “Understanding Sound Waves and How They Work,” https://science.howstuffworks.com/sound-info.htm#pt2, 2023, [Accessed 2025 Mar 7].

[37] J. Comeau, “Lets Learn About Waveforms,” https://pudding.cool/2018/02/waveforms/, 2024, [Accessed 2025 Feb 21].

[38] M. Mysteries, “Sine Wave,” https://mathematicalmysteries.org/sine-wave/, [Accessed 2025 Feb 23].

[39] W. Roberts, “Fourier Series,” https://w1.mtsu.edu/faculty/wroberts/teaching/fourier_4.php, [Accessed 2025 Feb 23].

[40] G. Reid, “Introduction to Additive Synthesis,” https://www.soundonsound.com/techniques/introduction-additive-synthesis, [Accessed 2025 Feb 20].

[41] E. W. Weisstein, “Fourier series,” https://mathworld.wolfram.com/FourierSeries.html, 2024, [Accessed 2025 May 16].

[42] ——, “Sawtooth Wave,” https://mathworld.wolfram.com/SawtoothWave.html, [Accessed 2025 Feb 23].

[43] “MoSCoW Prioritization,” https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html, [Accessed 2025 Mar 19].