University of Southampton

Faculty of Arts and Humanities

School of Music and Web Science Institute

Micro Music

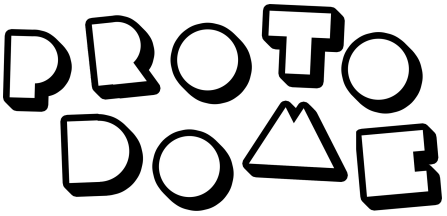
Exploring the Idiosyncratic Compositional Strategies Encountered in 1-Bit, Limited Memory Environments

Blake 'PROTODOME' Troise

ORCID ID 0000-0002-6539-4253

Thesis for the degree of Doctor of Philosophy

July 2020 – Version 1.3



UNIVERSITY OF
Southampton

Written by Blake 'PROTODOME' Troise. Compiled using the MacTEX-2019 distribution.

A significant part of this submission is the companion media and code artefacts. Whilst referenced, these are largely not included in this document and can be viewed at: `https://doi.org/10.5258/SOTON/D1387`. Additional materials can be found at: `https://github.com/protodomemusic/mmml` and `https://protodome.bandcamp.com/album/4000ad` and any queries can be sent to `hello@protodome.com`.

Research Thesis:
Declaration of Authorship

Name:

Blake Troise

Title of thesis:

Micro Music: Exploring The Idiosyncratic Compositional Strategies Encountered in 1-Bit, Limited Memory Environments

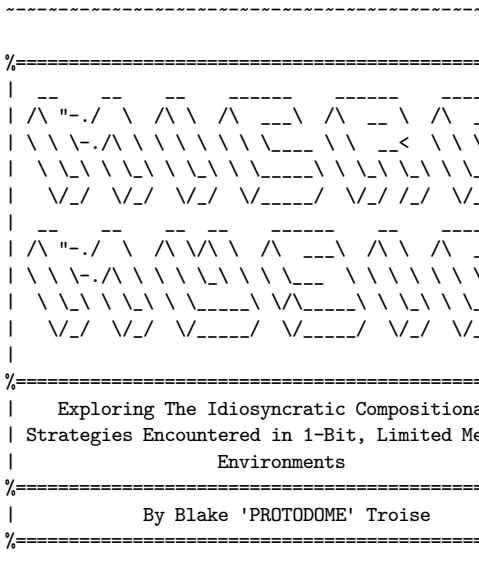
I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. None of this work was published before submission, but an adaptation of sections 2.1.1, 2.2, 2.2.2 and 2.2.3 has subsequently been included in the *Journal of Sound and Music in Games* (JSMG), published by University of California Press. Additionally, a public release of the code has been published on my GitHub (`https://github.com/protodomemusic/mmml`), alongside a public release of some of the musical materials (`https://protodome.bandcamp.com/album/4000AD`).

Signature:

Date:



1-bit music, generally considered a sub-division of chiptune, is the music of a single square wave. The only sonic operations possible in a 1-bit environment are amplitude and time, where amplitude is quantised to two, binary states: high or low, on or off. There are surprising techniques and auditory tricks unique to 1-bit practice: through layers of modulation, abstraction and clever writing, compositional methods can generate music far more complex than the medium would, at first impressions, indicate.

This is an environment defined by extreme limitation. Yet, belying these restrictions, there is a surprisingly expressive instrumental and compositional versatility. This research explores the theory behind the unique compositional techniques available to composer-programmer in low-memory environments; those which are essential when designing music to be as small as possible. These methodologies are considered in respect to their compositional implications and applications.

Using the 8-bit AVR microcontroller range to enforce both absolute memory restriction and hardware simplicity, and through custom synthesis software and algorithms, I have created a series of musical investigations focused on the central research question: using 1-bit synthesis, how does constructing music within one kilobyte (or a similarly limited memory environment) inform or change compositional methodology?

LIST OF ACCOMPANYING MATERIALS

MICRO MUSIC BOXES:

Two physical hardware synthesisers have been included as a companion to this commentary. Embedded on one is the primary µMML pieces (as they are named and structured in the 4000AD album release, found here: https://protodome.bandcamp.com/album/4000ad) and, on another, a selection of bitbeat compositions.

A guide of operation can be found in the media library listed below. Please read this before use.

MEDIA LIBRARY:

The companion online media/code repository is an aggregation of various examples and materials that support the research's written commentary.

This can be accessed by visiting:

WEBSITE	https://doi.org/10.5258/SOTON/D1387

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Overview and Research Questions

This document is a companion to the electronic 'music boxes' sitting beside you. It supports, and comprehensively explains, a portfolio of over an hour of 1-bit compositions, represented by nearly 40 kilobytes of program memory (roughly 400 times smaller than this PDF document). With this project I have set out to compose the tiniest music I can using the most minimal instrumentational environment possible. This project is an exploration of conceptual purism: a personal exploration of the compositional voice in an increasingly restricted environment. The danger with this process however, is aptly communicated by the following[1]:

> *I thought using loops was cheating, so I programmed my own samples. I then thought using samples was cheating, so I recorded real drums. I then thought that programming it was cheating, so I learned to play drums for real. I then thought using bought drums was cheating, so I learned to make my own. I then thought using pre-made skins was cheating too, so I grew my own goat from a baby goat. I also think that is cheating, but I'm not sure where to go from here. I haven't made much music lately, what with the goat farming and all.*

Even so, the most compelling thing about this process is, in my experience, once you start distilling your composition and stripping away superfluous components to produce musically dense micro-constructions, you realise that you can *keep going*. The attraction of embedded music, music hard-coded for a particular computational platform, is that the composer can control all elements of the composition, its encapsulation and its environment. How many composers can say that they constructed the platform, the synthesis, the instruments, the score *and* the composition? The sonic landscape of microcontroller music is dictated entirely by its creator. Using the 8-bit AVR microcontroller range to enforce both absolute memory restriction and hardware simplicity, and through custom synthesis software and algorithms, I have created a series of musical investigations focused on the central research question:

- Using 1-bit synthesis, how does constructing music within one kilobyte (or a similarly limited memory environment) inform or change compositional methodology?

Furthermore, the following two questions arise in sympathy:

- Is it actually possible to create complex, engaging music in less than a kilobyte of program space using 1-bit synthesis? (Including generation and synthesis code of course; no hiding behind decoders and interpreters!)
- What is the most economical method of creating 1-bit complex musical compositions in relation to memory footprint?

By the end of this commentary, the reader will understand:

[1] Often cited as an image macro, earliest example I could find and possible source here: `http://www.mnml.nl/phpBB3/viewtopic.php?f=17&t=62658&start=16`

- How this project fits within a wider context: the history of *1-bit music* and the *chipmusic* scene.
- The sonic and musical properties of a 1-bit waveform.
- How a 1-bit waveform might be synthesised in software.
- Approaches to organise and structure 1-bit voices musically.
- Potential compositional solutions to employ in low memory environments.

1.2 Project Motivations and Rationale

Somewhat less cogent than the previous research questions, I initially approached this project with two questions: "What cool music can I make using 1-bit synthesis?" and "In just how few software instructions can I make it?". Over the course of the project however, these questions evolved and the criterion for success has become ever more ambitious. This disposition is entirely reminiscent of the demoscene: the modem-networked, artistic coding community producing audiovisual artefacts one might consider somewhere between "poetry and graffiti" [1]. The specific focus on the *sizecoding* [2] ethos in my endeavours is a common motivation in many demoscene sub-cultures, where coding skill and proficiency is the primary currency in a cyberpunk meritocracy [3][2]. One can see recurrent evidence of this in the classification of events in *demoparties* (gatherings of 'sceners' often dedicated to competitive demonstrations of coding expertise), where competitions are split into computing platform and code size categories [4]. The miniaturist form correlates with my own enterprise; the ever smaller instruction spaces in which to express my musical language is incitement for inspiration. It is also consonant with demoscene practice that my process, as an inexperienced programmer, has often been trial and error, rather than adhering to rigorous mathematical or algorithmic models [1]. Essentially, elements of my approach to software design has been somewhat of a hacker/enthusiast.

On the nature of the project's methodology, it is useful to draw a comparison in literature. There is a limit to the rate of compression of a series of events before there are no longer coherent symbols to parse. This is best demonstrated with a simple thought experiment: decrement the maximum word count of a written task until only a single character remains. For example:

[2]Although there are certainly other contributing factors, technical prowess is a significant variable of success in a demoparty. One can often make submissions anonymously which are judged and ranked by a panel of other scene members (not always though and, where this is not the case, one could make a strong case that coding talent is *not* the primary factor of success.). Achievement is largely based on the submitted artefact, suggesting coding talent, rather than existing rank or status within the scene. This extends to chipmusic and similar online cultures *somewhat*, but one is never truly anonymous in any community and other traits, those real, or perceived to be real by peers, do have an additional impact on success. Recently, there has been some resistence to the idea of pure meritocracy in chiptune, perhaps because the scene is becoming increasingly diverse and inequalities more apparent.

```
I think, therefore I am
I am thinking, so I am
I think, so I must be
I am because I think
I've thoughts: I am
I think so I exist
I think thus I am
Cogito, ergo sum
I think so I am
I think ∴ I am
I think, I am
Think ∴ I am
Think: I am
I am: I be
Thk ∴ I'm
I'm real
I exist
I ∴ am
I ∴ I
I am
I'm
Me
I
```

The forms and techniques required to communicate an intelligible message change as the operable ceiling descends. At a certain point, a desired message is irrevocably altered away from its original intention; concessions and alternate strategies must be devised so that enough meaning is retained. One might find an emergent, unexpected beauty in this reduction. In a computational architecture, this boundary is more transient and indeterminate than it might be with text as there are various schemas and methodologies by which data can be stored and interpreted. My primary fascination, and the goal of this investigation, is to examine what happens to musical composition as the maximum semiotic resolution decreases. To return to the thought experiment, one might achieve retention of meaning through employment of linguistic strategies such as synonyms, rephrasing, concatenation and truncation. In a compositional environment, it is the musical analogues for these strategies I am most interested in developing.

If compositional reductionism is of most interest to me, why frame the project around 1-bit music? 1-bit music is certainly not required to explore compositional miniaturism, however it is an appropriate companion to the process. The role of 1-bit music in this project is to provide a simple, easily implementable set of instrumental forces with clearly defined parameters. Moreover, the relationship between composition and instrumentation is reciprocal and, in places, inextricably linked. As is mentioned in 2.2, 1-bit music is surprisingly expressive for a simple concept. As the research opts to keep both software *and* hardware as simple as is reasonably practicable, 1-bit synthesis has proven itself to be the best solution: it is exceedingly easy to implement in both software and hardware, and almost no additional electronic components are required. Additionally, practically

every microcontroller and processor on the market today (and since the dawn of electronic computing, see 2.1.1) can generate a competent 1-bit signal, fit for audio playback. Moreover, as the project is reductionistic and ruthlessly utilitarian in principle, software routines that generate 1-bit signals can be expressed in, some cases, *very* few instructions, resulting in minimal memory footprints. I recognise that not all instrumental, nor software, platforms will share exactly the same compositional solutions as this project, however I consider this admissible. Many of the solutions I have explored will be transferable and, where they are not, should provide a unique insight in to one (my) approach of writing miniaturist compositions. The same constraints may be placed on piano composition (for instance) but the operational boundaries are more ephemeral and arbitrary than composing within an enforced restriction, such as a microcontroller. Additionally, instruments with extensive techniques for sonic nuance and instrumental expression may have innumerable methods of composition; one could write expansively on different modifications to piano timbre and how this might affect compositional approaches. The limited variables involved in 1-bit synthesis constrains this freedom.

There is a subtle difference between compositional and instrumental techniques, although there is often overlap. For example, the classic chipmusic 'super-fast' arpeggio is more instrumental than compositional: if one were not limited by instrumentation (often, in the case of chipmusic, where there are few monophonic voices), an arpeggio might be replaced by a chord of simultaneous tones. I am more interested in the compositional *function* of such a technique; e.g. if an arpeggio is required, how might it be employed so that it contributes a minimal increase in program memory and effectively communicates musical intention. There are *many* characteristically chipmusic instrumental tricks that I am, and wider musical culture is, familiar with [5, 6]. The peculiar 'hacks' and eccentric workarounds devised to produce particular instrumental effects are one of the defining features of the culture [7]. Therefore, whilst I started this research (and my previous chiptune experience) on the *Nintendo Entertainment System* (NES), it quickly became evident that I was possibly unoriginal in my musical approaches to the hardware. What I could potentially say using the NES' paradigmatic dialect had largely already been said [8]. As 1-bit music relies on pulse waves[3], there is a significant overlap of instrumental techniques between not just the NES, but also the *Commodore 64*, *Game Boy*, *Atari 2600* and the *ZX Spectrum*; all popular platforms for chipmusic with vast bodies of existing musical works. Thus, where the project differs from the wider chipmusic 'library', is in the compositional construction. Keeping within the minimalist framework, and influenced by the accomplished beeper routines of the ZX Spectrum coding scene, this investigation looks for ways to find unique compositional ideas within the restrictions of the AVR platform.

A similar approach has previously been taken by composer Tristian Perich [9], producing forty minutes of 1-bit material on an *Attiny85* microcontroller (an integrated circuit (IC) with eight kilobytes of program storage space [10]). It *is* impressive, but with knowledge of the numerous techniques and approaches established in wider chiptune practice, I feel that there are many aspects that can be improved on[4]. In my doctoral work, the Attiny platform has deliberately been chosen for the extremely limited program memory available; a mere

[3] See 2.2

[4] Not that Perich was striving for the same goal as myself. As an exploration of 1-bit textures, *1-bit Symphony* it is indubitably entirely successful!

kilobyte of data for the *Attiny13*. The immutable headroom is a deliberate decision to enforce strict memory limits and counteract potential 'feature creep', where one might be tempted to "just fit in one more tiny function", bloating the code and undermining the investigation's premise. Additionally, if using lower CPU frequencies, the microcontroller can be run on 1.5 volts[5]: the voltage provided by a single AAA battery. The extremely low power requirement forces slower CPU speeds, thus encouraging more optimised coding strategies. This engenders similar effects to restricted program space and limits the synthesis to the characteristic 1-bit aesthetic[6].

The 8-bit AVR microcontroller series has another advantage: they are remarkably cost-effective. Whilst low-cost solutions are of no real consequence to the composition, accessibility in music is democratising. Attiny music (when delivered in a format similar to this submission's companion electronic 'music boxes') may be the cheapest electronic music one could possibly consume[7]: no prerequisite hardware such as computers, phones, or other media players are required. In fact, one could argue that the Attiny13 is one of the most affordable digital musical *instruments*; the embodied nature of the platform cannot be separated from the music in its psychological effect. For this reason any resultant artefacts of the research will be distributed physically, programmed for the hardware. Digital recordings seem periphrastic in comparison: literal expansions of program data orders of magnitude larger in size. In regards to possible backwards compatibility, although the primary μMML routine (see 3.2) runs at a blistering 8MHz, with a bit of tweaking (and reprogramming in AGC assembler), the whole project could be comfortably stored on, and potentially executed by, the Apollo 11 Guidance Computer[8] [11]!

I choose the term 'instrument' to describe the Attiny13 (or, more generally, the microcontroller) due to two main analogues I see with 'traditional' musical instruments. Firstly, the ability of the human composer to realise their compositional intentionality on the device's resultant output (the chosen purpose and function of the device is to create music) and, secondly, the fact that this compositional intentionality is not always realised *perfectly*; that there is some sonic stochasticity. For example, the trombonist can only guide the instrument into approximating their intention, the instrument's output has a degree of randomness. I am not entirely convinced by the definition of a musical instrument being something that only interprets, or approaches, absolute compositional ideas, as this definition may exclude a potentially 'perfect' instrument: one which can realise the ideas of the composer completely accurately. Even so, with most extant musical instruments, there does exist a degree of separation between the brain, and agency, of the performer, from the instrument's resultant vibrations. Of course, the Attiny here functions more like a player piano, in that it is both an instrument and its own performer. One must use caution when using the term 'performer', as it implies agency; a microcontroller cannot have intentionality, as this would require autonomy. The microcontroller will diligently enact whatever it has

[5] 1.8V officially [10] but, in the most unscientific of processes, I tried powering boards with a single AAA battery and it worked, very well too. An AAA battery should power an Attiny13 operating at 4MHz continuously for days! With consumption *this low* possible expansions might even see solar powered solutions.

[6] Higher speeds can allow for alternate, more 'hi-fi' strategies, somewhat undesirable for the research. See section 'Sonic Fundamentals' and 'Temporal Subdivisions'.

[7] As of writing, one can source the parts needed to build a 'Micro Music Box' (see `https://doi.org/10.5258/SOTON/D1387`) for certainly no more than £10.

[8] Assuming a clock speed of just over 1MHz, 36KB of "core rope" memory and 2KB of RAM. I realise that the IPS might not be up to it though...

been programmed to do, with no interpretation. Nevertheless, the 1-bit output of the microcontroller (in this instance) is not always identical, it will very slightly fluctuate based on environment, internal voltages and timing errors. If I roll a ball, covered in paint, down a hill, is the Earth (by means of gravity) the artist? Or can that be attributed to the ball? The microcontroller is an entity, acting freely from external input, that *appears* to have its own interpretation of the compositions it recreates. I feel that, pragmatically, *it is* a performer because it approximates one.

As it is not wholly in the spirit of the investigation, the research questions deliberately do not refer to data compression of files, or use of compressed audio file formats[9]. Reducing a file's footprint in flash memory by means of either minimizing the quality of the recording (lower bit and sample rates), or utilising archive formats and techniques, does not radically change the composer's enterprise. Or, at least, this does change the composer's practice in ways that satisfactorily answer the primary research concern of exploring how working in a 1-bit and low memory environment inform or change compositional methodology. I am interested in finding the limits of music when composed and stored semiotically; a string of symbols representative of atomised musical concepts (Kolmogorov encoded [12]), not merely storage for waveform data. Creation of this information adheres to almost none of the restrictions explored throughout the project as the composer is permitted to use almost any known method of generating sound data. Investigation into this problem would be that of the computer sciences, not music. It should be mentioned that, whilst the project is intimately related to technology, the motivation is always in developing artistic and compositional process, shaped by the code and platform architecture. Due to the subjective nature of this enquiry the research questions can only be addressed through a synthesis of deconstruction of existing materials, practice-led compositional exploration and physical implementation.

Computationally limited implementations of 1-bit music have had a comparatively brief period of time to develop and explore technique compared to other instrumental models, such as the orchestra or the classic rock outfit. 1-bit sonics have only been commercially relevant for maybe a decade before relegation to obsolescence and relative obscurity. Investigating and introducing this unusual methodology into modern musical practice may yield original results. Inversely, approaching legacy 1-bit technique from the perspective of a 21st century composer, influenced by the online 'chipscene', might have also affected results. I intend for this work to serve as an introduction to those who have not yet encountered 1-bit musical practice and, perhaps most importantly, serve as a comprehensive guide to composers who wish to create their own 1-bit music - irrespective of chosen platform. I have learnt a lot from the philanthropic efforts of online, anonymous 1-bit enthusiasts such as 'utz' and Alex 'Shiru' Semenov, both of whom have performed crucial roles in the documentation and propagation of 1-bit music as well as also being talented musicians and coders. I certainly hope that this investigation has furthered their efforts in the aggregation, engagement and expansion of the existing online literature, as well creating some exciting new sounds in the process.

[9]From my experience, 1KB is too small to store most common audio file formats, let alone additional code to decode an MP3, or uncompress a ZIP file, but the premise holds true nevertheless.

2 1-Bit Theory

2.1 Context and Culture

As evident from the *chipmusic* scene, it is an understatement to say that there is **a lot** you can do with a simple square wave. 1-bit music, generally considered a sub-division of chipmusic [13], takes this one step further; it is the music of a **single** square wave. The only operation possible in a 1-bit environment is the variation of amplitude over time, where amplitude is quantized to two states: high or low, on or off. As such, it may seem intuitively impossible to achieve traditionally simple musical operations such as polyphony and dynamic control within a 1-bit environment. Despite these restrictions, the unique techniques and auditory tricks of contemporary 1-bit practice exploit limits of human perception. Through layers of modulation, abstraction and perspicacious writing, these compositional methods generate music far more complex than the medium might, at first impressions, suggest.

Even if not originally conceived through ludic platforms (one can hear simple examples of 1-bit sonics in microwave interfaces and smoke alarms!) 1-bit music has, as it is understood today, been developed and propagated through video games and the companion demoscene culture [14]. Where systems such as the ZX Spectrum and early desktop computers contained severely limited audio capabilities, developers found creative solutions to do the seemingly impossible; polyphony, timbral variation and alterable volume all using a single, monophonic square wave. These tricks (often born through necessity) have established a broad and expressive 1-bit instrumental idiolect to rival that of any acoustic instrument: the idiosyncratic elements of its instrumental language that distinguishes 1-bit sonics from other instruments or platforms. In this chapter, Section 2.1.2 aims to place this practice in its historical and cultural context, and Section 2.2 outlines the theoretical basis of 1-bit audio and explores the unique instrumental capabilities of the 1-bit instrument.

2.1.1 A Short History of 1-Bit Music

The history of 1-bit music is inexorably linked to the history of computational music; 1-bit synthesis often presents itself as the simplest solution when generating digital audio (see Section 2.2). The earliest examples of computationally synthesized music emerge with the advent of programmable, electronic computers in the post-war era of the 20th century[10]. Consequently, the first recorded instance of a digital composition was a program written for the BINAC computer in 1949 by Frances 'Betty'Holberton (then Betty Snyder). Both Jack Copeland and Jason Long's claims [16], and 1-Bit Forum user utz's (independent) research [17], make a very compelling case for this assertion which changes the origin of computer music from the often cited CSIRAC and Manchester Computer musical software of 1951, to two years prior [18, 19, 20]. The very first sequenced music for an electronic computer seems to have been Holberton's rendition of *For He's a Jolly Good Fellow* to celebrate the completion of the BINAC project [21]. In the 1950s, Alan Turing outlined the theoretical basis for computational synthesis in the Programmers'Handbook for Manchester Electronic Computer Mark II where an audio routine is proposed [22]. The described application for generating sound is pragmatic, rather than creative: Turing's "Hooter" function generates tones, clicks and pulses as a feedback system for computer-human interaction. The document suggests that the generated tones might allow a computer's operator to "listen in"to the progress of a routine, somewhat similar in function to later technologies, such as the dial-up modem. Despite Turing's initial intentions, this routine was eventually employed to create the earliest surviving computer music: a monophonic rendition of *God Save The Queen* [23][11]. Both the BINAC routine and the "Hooter"function would, most likely, have used 1-bit synthesis; the method of generation described by BINAC engineer, Herman Lukoff, suggests a process similar to contemporary 1-bit routines: *'... by programming the right number of cycles, a predictable tone could be produced. So BINAC was outfitted with a loudspeaker⋯'* [16]. Additionally, Turing's "Hooter" employed a series of clicks, which suggests a method similar to the very thin pulse widths used in ZX Spectrum sound routines (see section

[10]Contrastingly, the earliest citation I could find of computational music is much older: a speculative musing by Ada Lovelace in the early nineteenth century —a century ahead of its actualization [15]

[11]I feel the selection of 'God Save The Queen' by the Mark II engineers for the BBC journalists has two primary functions: firstly, this is a patriotic demonstration of British technological prowess (perhaps with the subtext of dominance) in a post-war era. Secondly, the song choice is an obvious consequence of Technique 4 in Section 4.3, especially when considering the two companion pieces: *Baa Baa Black Sheep* and Glen Miller's *In The Mood*. All three would be well known to the British population of the early 1950's and have clear melodies. Crucially, monophony can be used to communicate these pieces coherently, without the need for further harmonic context via arpeggios, expanded melodic writing or additional channels. Additionally, although modern computing has its genesis in the wartime militaries of Western powers, I'm not sure that this sentiment has carried over to the modern chiptune scene. There are certain demographics that dominate the compositional and cultural landscape of the current scene, but these are somewhat detached from this origin. Chiptune is largely a Western phenomena (including Japan) and largely male. One might argue that this is because of computing's origins, but I think it has more to do with the home console/ home computing culture of the 1980's. This said, it does still raise the question, to what extent has computer music been shaped by historical statements of power, politics and religion? Although examples of protest and politically charged musics perhaps more overtly address this question, I think the most interesting examples in chiptune are perhaps more subtle. For example, those chipmusicians from old member states of the USSR have a different set of computational platforms that they grew up with and, thus, have entered chipmusic from a different angle. Similarly, the traditional Japanese/Western divide in chiptune compositional style extends beyond chiptune itself and is reflective of the differing cultures (broadly, Japanese musicians generally focused on melody and counterpoint where Western composition was concerned with extended techniques such as super-fast arpeggios). My work does not attempt to actively engage with these larger, socio-political topics — not because they are not relevant to the discussion — but because it is beyond the scope of both the project and my ability. Open-source cultures will always be political in themselves in that they negate geographical and state divides making them, to some extent, anti-hierarchical.

Timbre)[12]. There are numerous subsequent examples in the 1950s of music created using research and military computers, the majority offering similar, monophonic lines of melody expressed using square waves [24, 25, 26]. For example, in August 1951, Geoff Hill wrote a 1-bit routine for the CSIR Mk1 which was performed publicly at the Conference of Automatic Computing Machines in Sydney [27]; in 1955, Norman Hardy and Ted Ross rewired a console control lamp to a speaker to perform a rendition of *Bach's Partita No. 3 in E major BM 701* [28]; and, from 1958, service technicians wrote music programs for the German Zuse Z22 computers, one of which was even distributed officially by Zuse [29].

In the 1960s and 1970s, as technology became both faster and more accessible, programmers began to experiment with alternative approaches. An EP, released in 1970, was created using the DATASAAB D21 and D22 [30]. These recordings demonstrate the use of super-fast arpeggios to simulate polyphony (alongside other, perhaps more advanced, audio solutions that were not strictly 1-bit) [29]. In 1970, Thomas Van Keuren, a programmer working for the US military in Vietnam, independently employed rapid arpeggiation, programmed on a UNIVAC 1050-II US military computer [31]. When home computers became readily available to hobbyists, during the late 1970s, the more complex and widespread routines begin to materialize. These routines explored more advanced techniques, such as true 1-bit polyphony [13], and heralded the age of the computer music enthusiast: a prelude to the subsequent chipmusic and demoscene cultures. In order to keep the manufacturing costs low, and maintain the affordability of home computers, functional concessions had to be made. Home computers did not have the memory capability to store large amounts of sampled data, thus alternative strategies were devised to include audio in software applications, most frequently video games [33]. Dedicated sound hardware utilized a wide variety of methods, such as frequency modulation and wavetable synthesis [34], however even these could be expensive. As an alternative, PCs were frequently shipped with internal speakers that were attached directly to a processor output pin [35]. Systems such as early models of the ZX Spectrum initially provided the programmer no alternative but to use 1-bit music [36], and those who did not wish to invest in a sound card for their desktop computer could still experience audio in games and software via the internal speaker [14]. These requirements encouraged games publishers to find the most interesting solutions possible to distinguish their product from the competition and garner more sales [42][15]. With this attitude of 'progression', the industry ultimately discarded 1-bit audio in favour of greater sonic capability and versatility. Development of 1-bit practice was then adopted by the hobbyist; those with nostalgia for their 1-bit platform, or those fascinated by the medium. The computers (such as the ZX Spectrum) that booted directly into a programming interface, commonly BASIC [43], allowed the casual user immediate access to functions that could beep and, with a little clever extrapolation, be coaxed into playing melodies. Those engaging in the emerging demoscene

[12] Or, perhaps, these were actually saw-tooth generators. It does seem unlikely due to the nature of binary, digital outputs (see section Fundamentals), but I could not find a definitive source documenting the actual synthesis methods used.

[13] Such as The Music System, published by Software Technology Corporation in 1977, which employs true 1-bit polyphony [32].

[14] There must have been a significant number of users without sound cards; many games were written to support the PC speaker. To name just a few: DOOM [37], Prince Of Persia [38], SimCity 2000 [39], and Total Eclipse [40]. The companion instruction booklet to the DOS game *Crime Wave* even has instructions on how to connect the PC speaker directly to a stereo system [41].

[15] Fritsch discusses this in respect to arcade systems, not home video game systems. There exists the societal attitude that each successive release of a technological platform/product is a 'progression'; striving for greater computing power, memory capacity or graphical 'realism'.

and chipmusic cultures, often having experimented with these simple routines as children or teenagers, pushed what was previously possible. They were aided by Internet collaboration, through competitions, communal sharing of code and a general enthusiasm for the medium.

2.1.2 The Chipmusic and Demoscene Culture

1-bit music is almost always associated with chipmusic[16]. The reason for this relationship is related to contemporary 1-bit practice and chipmusic's unwritten artistic 'manifesto'. Chipmusican Niamh "Chipzel" Houston perfectly summarises the idealogical change and aesthetic difference between chipmusic and historical computer music efforts. In a 2014 interview with VICE UK [44], she states:

> *Some might say... [we] decided to quit keeping up with technology sometime around 1999. But I'd argue that's symptomatic of the society we live in, where you'd sooner throw out the old than properly appreciate and celebrate its quirks.*

Houston is referring to the enjoyment of creating music by exploiting limited systems for their idiosyncrasies and unique stylistic qualities. This is the process of making music with legacy hardware for the joy of its aesthetic alone, as opposed to exploring the novelty of the compositional process, or pursuing more traditional, human performed musics in software. An article in the third volume of the 1980's *The Best of Creative Computing* magazine [45] demonstrates the historical eagerness to move away from "square wave" compositions and to higher fidelity, computer aided audio. These attitudes are a clear divider between the historical and modern contexts of chipmusic practice. Rather than possessing the 'space age' qualities of an unfamiliar sound-world, today the sound of 20th century computers and their associated programmable sound generator[17] paradigm have a retrofuturistic quality. In fact, the term chipmusic, or *chiptune*, has only ever referred to the genre retrospectively, entering the lexicon of Amiga musicians around 1990 [49]. Originally this referred to the tradition of making music to appropriate and approximate that of Commodore 64 musical practice. Carlsson therefore considers the genre as "permanently" retro, a sentiment that would seemingly exclude video game music, or music made for historically contemporary systems from the chiptune definition — by means of its intention. Brian Eno recognised this trend, predicting that the signature characteristics of a medium would be replicated and emulated as soon as they could be avoided [50]. It is perhaps not *replication*, but continuation and evolution that defines chipmusic today.

At its most fundamental, chipmusic refers to music either written for, or in the paradigm of, computational systems that do not rely on *redbook audio* (or similar, recorded formats) for musical reproduction [51, 7]. The exact definition of chiptune is a somewhat contentious issue amongst chipmusicians and fans; divisions occur in differing opinions as to whether hardware authenticity is required for a piece to be truly 'chip' [52, 53]. Whilst there may be disagreement, most sources are consonant in that there does exist some form of cohesive, electronic aesthetic born from early, hobbyist online communities [3].

Anders "Goto80" Carlsson attempts to address this issue by reducing chipmusic ontology

[16] There are some who may disagree; I can find no evidence of Perich discussing the relationship between his work, ZX Spectrum music and chipmusic. However, his work is occasionally mentioned in chiptune discussions and literature, for example in Leonard Paul's article *For the Love of Chiptune* [7]

[17] A programmable sound generator (PSG) is an integrated circuit (IC) with the ability to generate sound by synthesizing basic waveforms.[46, 47, 48]

to three of its most fundamental facets: medium, form and culture [51][18]. Chipmusic as a medium addresses the most prevalent interest of populist literature, the relationship between chipmusic and its parent hardware [54, 55, 56, 44]. These sources seek to define chipmusic through video gaming nostalgia [44], specific platforms and their individualistic array of miniature, embedded synthesiser 'sound chips'. The main impediment to accepting this approach *definitively* is that establishing exactly which hardware constitutes valid 'chiptune paraphernalia' is problematic. Most would argue that music written for the *Sega Mega Drive* (or *Sega Genesis*) is chipmusic [7, 57]; the Mega Drive had a *Yamaha YM2612* PSG capable of six monophonic FM channels with four operators per channel [58, 59]. Very similar chips were employed in a range of Yamaha keyboards of the time, for example the popular TX81Z rackmount synthesiser and DX21 keyboard[19] [60]. As most would argue that music made with Yamaha FM keyboards does not fit the criterion[20] [51], this strictly materialistic view cannot describe the phenomenon entirely.

Addressing chipmusic as *form* regards the defining characteristics of the 'genre' (another contentious topic, addressed below) as compositional language: the common, distinguishing musical voice of the practice. If one were to judge a piece of chipmusic on the artefact alone, irrespective of technology and surrounding culture (the latter *somewhat challenging* admittedly), it might be difficult for a listener to differentiate between music created with hardware and music made in *modern digital audio workstation* (DAW) software. Moreover, as Carlsson recognises, both analogue synthesisers and chiptune technology produce the same, fundamental waveforms [51], therefore the language of chipmusic must extend beyond its basic timbres and synthesis. Logically, chipmusic must also be defined by structural, organizational and architectural limitations. These constraints include, perhaps most notably, restricted polyphony and super fast arpeggiation [61], but also extend to dynamic ranges [7], tempi [62] metre [52] and nearly all facets of musical practice. A brief online search with the question "How to write 8-bit music" provides numerous sources addressing the compositional impact of limited voices in chipmusic [63, 64, 65]. Additionally, as traditional platforms were often limited by how frequently musical parameters, and PSG hardware registers, could be updated (for consoles such as the NES, this limit was often the television's vertical refresh rate), strict quantisation of musical events is a common property in hardware chiptune. Unquantised, humanised passages and extended polyphony often sound 'inauthentic', a lesser approximate to the chipmusic aesthetic. Nevertheless, many of these defining characteristics are found in other musical genres: creative use of limited polyphony in the small ensemble, hard quantisation in *electronic dance music* (EDM) and even super fast arpeggios in Bach's *Chaconne for Solo Violin*. As this is the case, seemingly the only definition one could ascribe to chipmusic is that there is no single, concise definition. The defining ingredients of chipmusic might be visualised as a Venn diagram; existing as an intersection of numerous properties.

To define chipmusic as culture may seem somewhat counter intuitive, especially as the genre appears nominatively deterministic. Speaking now as an actor within the scene, there has been a noticeable shift away from hardware and compositional authenticity (reflected in

[18] I have largely inhered my structure for this section from Anders Carlsson's perceptive interpretation of the scene, found at `https://chipflip.wordpress.com/chipmusic/`. I encourage the reader to read his words on the subject for themselves.

[19] Patches for both the Genesis and other Yamaha keyboards were even compatible [60].

[20] Bands such as *Level 42* and *Huey Lewis and the News* certainly aren't considered to be chiptune-fusion!

Carlsson's experiences [49]). Genres such as EDM and video game music (of any genre) can now comfortably fall under the increasingly nebulous chiptune umbrella — given the right framing. Conversely, it is interesting to me that Perich does not describe his music as chiptune, instead choosing "low-fi electronic". Whilst 1-bit Symphony could easily define its identity by chiptune and demoscene practice, Perich's decision to distance his work from the existing culture seems to reorient its appreciation. His audience approach his work more cerebrally, enjoying the medium for its unique instrumentation and aesthetics, seemingly free from the baggage of nostalgia and platform sentimentality [66, 9]. This is, sociologically, an interesting period in the genre's evolution and may indicate as to whether nostalgia and hardware re-appropriation was a significant part of chipmusic's original appeal, especially as both online and offline culture increasingly replaces the practice as the collective's primary adhesion. Younger musicians who encounter the idiosyncratic 'sound' of chipmusic without the nostalgic bias of childhood video game experiences may not find the same enjoyment and interest in the aesthetic. Born in 1990, growing up with DOS and Windows, the only platform I was familiar with that created the characteristic chipmusic 'bleeps' was the Game Boy, one of the last platforms to rely on music generation in this way. As Carlsson notes, hardware authenticity is of decreasing importance to the appreciation, or understanding, of the scene — perhaps this will indeed be the final decade of chipmusic as a distinct 'genre' [67]. My own generation may well be the last era where the ideals and attitudes of the first chiptune generation are still reflected. This may read melancholic, but is far from it. The scene is evolving and, if music made with old hardware is increasingly separated from 'game music' in the minds of both composers and audience, appreciation of the medium as a set of instrumental forces may lead to novel practice, free from the restraint of nostalgia, instrumental convention and musical pigeonholing. There has been some recent discussion over whether or not chipmusic is worse today as the core focus is shifting from 'technical innovation', or hardware purisim, to something diluted, videogame focused and commercialised. Personally, I think the chipscene is stronger than ever musically. This decade has seen some of the most innovative, technically competent releases yet; rivalling any other 'serious' genre.[21] In general, the chipscene is not as pretentious as this analysis — it somewhat kills the vibe! I think the unwritten rule is, don't worry about what is or is not chiptune, just write some music and have fun!

Returning to Houston's definition of the chipmusician's mentality [44], chiptune could be considered a musical counter-culture that, by embracing 'obsolete' hardware, is anti-mainstream through its apathy to commercialism. This is true to a certain extent: whilst repurposing discarded technology for artistic intentions is in direct opposition to commercial trends such as planned obsolescence, the further the hardware manufacture date is from the present, the more expensive it becomes to engage with as a 'hardware purist'. In fifty years time, due to the increased scarcity of working hardware[22], will chiptune, as it exists today, be recognisable? Perhaps the anti-commercial spirit of chiptune is not in the hardware itself, or even the music created, but the approach. The chiptune/demoscene communities share much in common with the current maker culture, which is perhaps where this project more

[21] I'm not going to put all my favourite releases inline, but if you are inclined, please do listen to the following: Chibitech's Moe Moe Kyunstep [68], FearOfDark's Motorway, Shnabubula's NES Jams, cTrix's A For Amiga and Seajeff's Magenta.

[22] Even though components will fail, there will be no shortage of shells and inoperable units: the Game Boy sold around 118.69 million units [69]!

closely aligns. Maker musicians have created music using a wide variety of materials and technologies, using, for example: 3D printing [70], floppy disk drives [71] and marbles [72]. Although these platforms are not traditionally considered chipmusic forces, the ethos and motivation certainly is. There will (hopefully) always be musicians curious to see technology coaxed into creating music and, even if compositions produced with these systems are not novel, the practice drives innovation. This is not necessarily anti-capitalist; often the latest hardware releases see hackers and musicians creating content using these systems[23] but through the subversion of popular, commercial, preconceived notions of function, comes authenticity.

So finally, by what musical taxonomy can chipmusic be understood? Whilst chipmusic is oft assigned *genre* status, I would argue (as have other chipmusicians [74, 75]) that it is better described as, and has more comparable attributes to, a medium, rather than a genre. Extending this sentiment, I feel similarly about 1-bit music; it seems to me that the medium's defining compositional characteristics are less arbitrary tropes, and more akin to the mechanical limitations of acoustic instruments. Although particular instruments prove more suited to certain genres (for either technical or associative rationale) the instrument is generally the medium by which genre characteristics can be expressed. Gifted with a large polyphony, variable volume, timbral flexibility and, if one considers the programmed nature of the music intrinsic to the instrumental capability, able to recite pitch information as fast as it can update a waveform, the 1-bit instrument can tackle most genres. Perhaps most intriguingly (in the pursuit of authenticity), it is writing in those styles, those in which it is hardest for 1-bit music to communicate concisely, its distinguishing characteristics become most evident.

So, can we consider 1-bit instrumentalism to be virtuosic in the same way we might a traditional instrumentalist? Because, through typical 1-bit music compositional methods, it is relatively simple to perform widely recognised displays of virtuosity (fast passages of notes, large leaps in octaves, gratuitous soloing, etc.) the nature of the virtuoso is communicated by the programmer, or composer's, intentionality. This is certainly not unique to chip, or 1-bit, music and is true of much computer music[24]. Novelty, and instrumental accomplishment, in 1-bit music (and chipmusic) is conveyed via two primary methods: authenticity and application. Authenticity refers to both platform (is it emulated or using real hardware) and intelligence of execution (does it use imaginative, original or unorthodox compositional or programming techniques) [52]. Application is more compositional: essentially, how well is the composition realised using the chosen soundworld? Originality of implementation is not required in the same way, only mastery of the 'standard techniques' (those which have been selected by process of memetic evolution and subsequently embedded in the musical vocabulary) and the demonstration of this proficiency through liberal, competent application[25]. Jaelyn "Chibitech" Nisperos' *Moe Moe Kyunstep* [68] is a perfect example of chiptune virtuosity. Whilst it is expertly composed, it is the instrumental proficiency, unconventional timbres and display of mastery over the hardware (in this case, the Nintendo

[23]For example, RoBKTA's album *SwitchTunes* [73], composed using a *Nintendo Switch*, a gaming console released in 2017.

[24]For example, ad hoc genres such as *Virtual Jazz*, proposed by digital musician 'Aivi'. See the discussion here: `https://twitter.com/waltzforluma/status/1159900481205878784`

[25]I am drawn to parallels in jazz, where ability to comfortably improvise over Coltrane's *Giant's Steps* expresses a similar sentiment [76].

Entertainment System) that makes the piece so celebrated in the chiptune community [77]. The importance of technical realisation in the appreciation of this work is clearly of significance to Nisperos, as she includes an overview of how the bass parts were created on the song's release page [68].

So, where does my own work for Attiny fit within this model? If platform were not important to the process, the compositional investigation could be completed on *any* system with a negligible change in aesthetic: perhaps on an emulator or recreation in a DAW. Obviously this is not the case, further supported by those whose primary interest in the practice is in the re-purposing of old video game consoles for musical application [54, 55, 56, 44]. The attraction of many to Perich's 1-Bit Symphony is the bespoke, real-time performance of work by the microcontroller itself [78, 7], as is, I feel, the appeal of my own work. Despite the elitism and hardware politics that emerge from the platform centric view of chipmusic, there is novelty to the performative aspect of musical electronics and the definition should encompass this to a certain extent. Where my own work and Perich's diverge is in both form and culture. Whilst 1-Bit Symphony is extraordinary in many ways, it fails to incorporate some of the elements that makes *classic* 1-bit chipmusic so compelling: pulse width sweeps, gratuitous pitch bends and faux ADSR enveloping, to name just a few.

2.2 Sonic Fundamentals

Instrumental technique in 1-bit music is shaped largely by its signal, perhaps to an extent that other instrumental and musical platforms may not be. To maximize expression and proficiency when using this environment, one must have a basic understanding of the theory and its implementation. The relationship and discrepancies between an ideal, logical pulse wave and its translation to analogue and acoustic domains can be exploited to musical effect. As such, practical application and exploitation of these concepts result in unique compositional techniques exclusive to 1-bit music, belying the nature of the environment. Appreciation of how the 1-bit waveform acts conceptually (and psychoacoustically) is necessary in understanding and implementing timbral and sonic interest in 1-bit music.

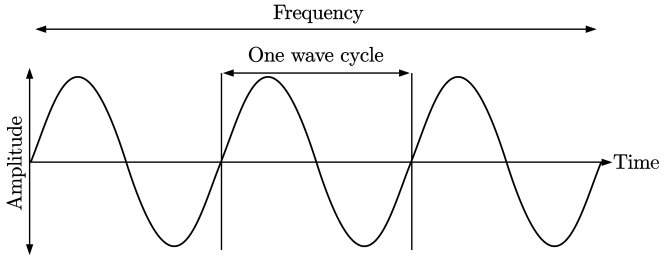


Figure 1: Oscilloscope view of a sine wave

Whilst the *sine wave* is the most fundamental sonic component of the acoustic domain [79, 80], in the digital world this is arguably the pulse wave. 1-bit music is the music of pulse waves: simplistic waveforms with binary amplitudinal resolution. Fundamentally, a waveform is the shape of an oscillation or vibration, moving through a medium, around a fixed point. A waveform's primary attributes are amplitude and frequency. Plotted on a two-dimensional plane, amplitude is variance on the Y axis and frequency is the addition of time on the X axis (Figure 1) [81]. Whilst a sine wave's amplitude can be traced continuously across the Y axis, in a 1-bit environment, for any given point in time, each step of the waveform can be only one of two states: logical high or low.

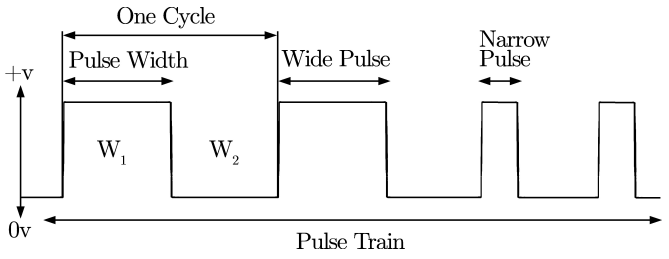


Figure 2: The topography of the pulse wave.

The pulse, or rectangle wave (described as such due to the geometry of the waveform's graphed appearance, see Figure 2), is a non-sinusodial periodic waveform defined by series of instantaneous switches between two distinct logic levels [82]. Often, these quanta are electronic oscillations between two voltages (usually positive and ground), however pulse waves can exist as a longitudinal pressure wave or as an abstract, mathematical function. The period between two amplitudinal events is considered a *pulse*. Regular periodicity of pulses (a *pulse train*), are known as its *frequency*: the rate of repetition in any periodic

quantity [83]. Discernible frequency is required for texture and pitch coherency, whereas random distributions of pulses result in unpitched audio noise, approximating white noise [84]. This definition is important as it suggests the first instrumental capability of the 1-bit pulse; the ability to produce both percussive (unpitched) and melodic (pitched) sonorities. The 1-bit musician must simply change the order of pulses from regular to unordered to generate two vastly different textures.

The duration of a pulse event's *mark* (high) time is referred to as its *pulse width* (W_1 in Figure 2; the *space* (low) time is indicated by W_2) [43]. The relationship between the pulse width and the total cycle duration can be expressed as either a ratio, or a percentage, known as the waveform's *duty cycle*. A duty cycle of 50% (a ratio of 1:1) would indicate a pulse width with equal mark and space time. Whilst there *is* technical differentiation between the definitions of 'pulse width' and 'duty cycle', both terms are often used synonymously in the chipmusic community, referring to the ratio between the waveform mark and space time [85]. Whilst theoretically infinite in variation, the maximum number of unique duty cycles is limited by both hardware fidelity and human inability to discern a waveform's *phase*[26]. A duty cycle of 75% (3:1) cannot be aurally distinguished between a duty cycle of 25% (1:3), as these are considered *phase inversions* of one another [86][27]. The mark time of the 75% waveform corresponds to the space of the 25% waveform and, when these are inverted, are perceptually identical (Figure 3). Due to this effect all similar widths above and below 50% are timbrally identical to the human ear (by means of: $50\% \pm n, n < 50\%$).

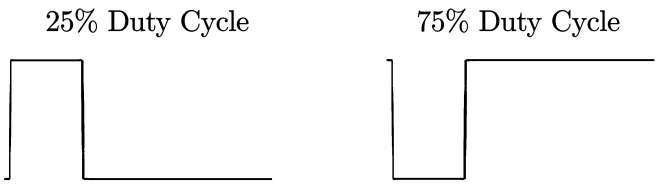


Figure 3: Duty cycles of ratios 1:3 and 3:1. Notice the inverse polarity of the mark and space time.

As a practical example of how this might implemented in software, Listing 1 (written in C pseudo-code) generates a single square wave, demonstrating perhaps the most elementary 1-bit tone generator and its composition. Here we can see a scripted version of the aforementioned list of basic operations required for tone generation. The desired frequency can be set with the `frequency` variable. The timer variable `pitch_counter` decrements continuously within the main loop of the program, checking if zero at each evocation. When this is valid, `pitch_counter` is set to the value of `frequency` to be compared against 0 indefinitely. This provides regularity and, when the `if` condition is true, I/O port logic level (hence voltage, hence amplitude) is alternated to create frequency. As long as `frequency` remains constant, the equal intervals between pin bitwise XOR operations produces a pulse wave of equal high and low durations: a square wave.

[26] Phase is the position of a point of time on a waveform cycle, subdivided into 360 degrees of possible offset from the origin [81].

[27] The waveforms are offset by 180 degrees.

```
if(pitch_counter-- == 0)
{
  pitch_counter = frequency;
  output ^= 1;
}
```

Listing 1: A square wave generator written in C pseudo-code. Note that the `pitch_counter` and `frequency` variables actually represent *half* the period; as decrementing to zero changes the amplitude, two complete cycles are required to generate a complete waveform.

2.2.1 The Effects of Practical Implementation

Discussion and analysis of aural techniques in reference to 1-bit theory *alone* excludes the dependencies between the theoretical and actual waveform; the 1-bit paradigm does not perfectly translate from the conceptual to the acoustic domain. The digital to analogue converter (DAC) and electronic circuitry can misrepresent signals, subjecting pulse waves to deformations such as *ringing*, *rounded leading edges* and *overshoot* [87], demonstrated in Figure 4. Even internally, a microcontroller may not update all bits in a register at the same time, causing I/O pins to momentarily output an erroneous value [88]. These distortions alter the intensity and weighting of the signal's harmonics: component sinusoidal tones arranged in successive integer multiples of the first harmonic (known as the fundamental) [89, 79][28]. This arrangement is known as the *harmonic series* and is responsible for the perception of timbre: the identifying characteristics of a sound.

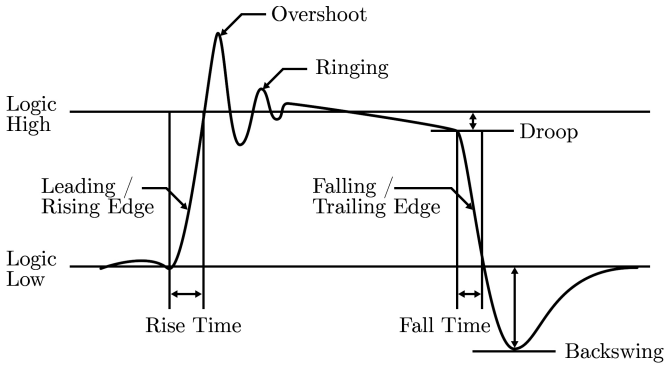


Figure 4: A diagram of possible distortions and deformations to the pulse component of a rectangle wave. Ringing is oscillation after the leading edge, often proceeded by overshoot, where the signal's amplitude increases beyond the logical high level. These distortions will indicate that the signal's harmonic image deviates from that of an ideal pulse wave. [89, 90]

Figure 5 illustrates relationships between timbre and waveform. The first waveform demonstrates the 'ideal' square wave[29], with a (comparatively) excellent frequency response, as indicated by the intensity of the dark bands in the *spectrogram* view [90]. A spectrogram is a visual representation of the harmonic components of a sound. The pictured bands represent individual harmonics, where the lowest band is the fundamental. The effect of rounded leading edges to the second waveform has a very obvious timbral effect when compared to the ideal waveform's spectrogram image. Aurally, this will 'dull' or 'muffle' the sound, as if heard through fabric or a wall. The third shows a deficiency of the lower frequencies,

[28] Or, alternatively, deficiencies in the frequency spectrum may result in an alteration of the waveform.

[29] It should be mentioned that the term *square wave* refers specifically to a pulse wave with a duty cycle of 50%

which will have 'brighter', yet 'thinner' quality due to a weaker bass response than the other two examples. Therefore, when approaching practical 1-bit composition, one must consider the method of physical implementation by which the listener will experience the composition. Ultimately, any software generation routine will be subject to timbral alteration by the physical method of sonic propagation, including electronics enclosures. As a practical example, Tim Follin's work for the ZX Spectrum game *Chronos* does not account for this, sounding coherent in emulation but unintelligible on some systems[30].

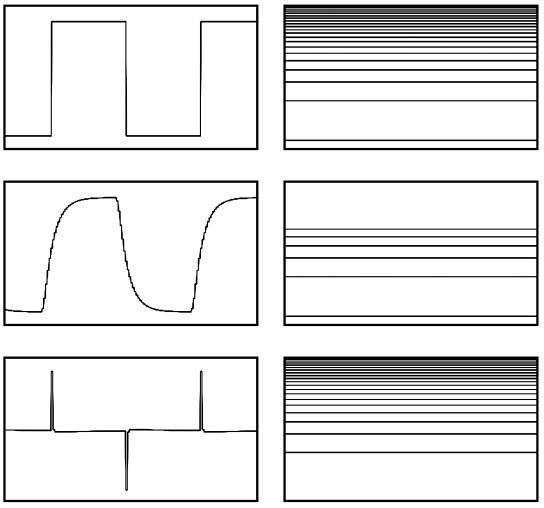


Figure 5: The waveform (left) and frequency spectrum (right) view of three permutations of square wave over time. From top to bottom: excellent response, poor low frequency response, poor high frequency response. The scaling of the spectrogram is logarithmic; skewed to align closer with human perception of pitch and severely thresholded to the loudest harmonics to clearly demonstrate the relationship between waveform and timbre [92]. Generated at a sampling rate of 44100Hz with Image Line's *3x Osc* [93].

Despite this, variations of wave shape away from the mathematical ideal each have an individual aesthetic appeal, some alterations more preferable than others. The *Game Boy Sound Comparison*, written by chipmusic composer Herbert Weixelbaum, is an investigation juxtaposing the discrepancies between square waves generated by ten different Nintendo handheld games consoles [85]. All consoles are running the popular Game Boy software, *Little Sound DJ*. Weixelbaum produces a square wave of identical duty cycle, amplitude and frequency in software, then appraises the strengths and weaknesses of the hardware, in respect to the resultant signal. The aesthetic importance Weixelbaum places on his chosen analogue, the original Game Boy, is interesting as it is not the most perfect replication of the square wave. In fact, it will have a poorer bass response than the Nintendo DS running the same software, for instance. These micro variations in timbre become a significant instrumental concern for the chipmusican. The sheer ubiquity of the Game Boy *pro-sound* modification (by-passing the hardware amplifier to obtain a more desirable signal) is evidence to this [94, 95]. These examples highlight the aesthetic consequences analogue 'malformations' can have on timbral qualities external to the software ideal environment.

Since this project utilizes Atmel's AVR Attiny series of microcontroller [10], a similar process to Weixelbaum's was undertaken. To identify the step response (and thus timbral characteristics) of the Attiny13, the raw output was analysed via oscilloscope, demonstrated

[30] This is humorously observed in a 1987 review in *Crash* magazine, describing the music as "a strange bit of title sound (rather than music)" [91].

in Figure 6[31]. The signal was sampled from a single I/O pin, with only a few intermediate components, to achieve as accurate a representation as possible[32]. The microcontroller was flashed with the `square.c` program and clocked at roughly 4MHz. The investigation originally examined the output of an Attiny13, Attiny45, Attiny85 and Atmega168, however all had identical results with the equipment used. There may well be minute variations caused by the different architectures but, for this study, if the signal both sounds and acts the same, there is functionally no difference.

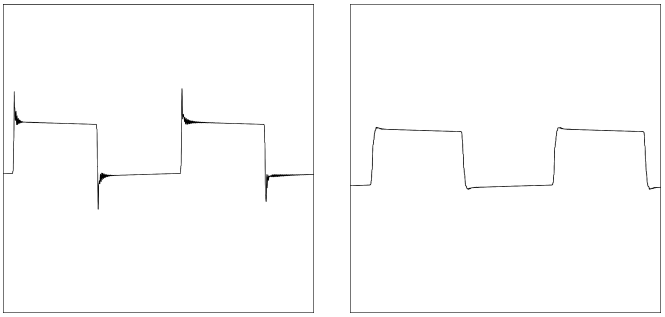


Figure 6: The resultant, graphed artefact of the `square.c` program (`https://doi.org/10.5258/SOTON/D1387`). Sampled from an Attiny13 at 44100Hz. The left-hand waveform is largely unfiltered; it was recorded with only a 10kΩ in series from the I/O pin. The right-hand waveform has a simple RC filter (passive low pass), using a 2.2μf capacitor, to smooth the overshoot and wobble.

The waveform to the left represents the unaltered output. It is subject to fairly extreme overshooting and, subsequent, ringing, as seen in the transitional periods of the individual pulses. This could be a product of parasitic capacitance/inductance in the system, or perhaps a 'glitching' of the output voltage [88]: as mentioned previously, digital data may take time (microseconds in duration) to stabilise at the correct, internal levels. Either way, this extra distortion to the signal manifests itself as an increase of intensity in higher harmonics. The right-hand waveform does not have this erroneous signal. The voltage wobble has been corrected with a simple RC low pass circuit [96]. The reader may also notice a slight 'rounding' to the waveform's leading edge, similar to the second example in Figure 5. The RC circuit works on this very same principle, performing a reduction in the higher harmonics to return the frequency spectrum to the ideal, thus 'smoothing' the rising edge of the waveform. The companion *simple square*[33] video demonstrates the busy character of the ringing as well as the sonic differences between the filtered and unfiltered waveforms.

As part of the investigation, I wrote a simple program (`1-bit-generator.c`[34]) that builds wave files from the output microcontroller code in software, rather than synthesised in real-time by the hardware. This allowed me to investigate some of the concepts (and create visualisations) with a digital representation of a 1-bit signal. The comparison between the output of this program and the actual hardware is significant: timing errors in the code are ignored by `1-bit-generator.c`, but colour the sound of the microcontroller analogue with crackles, pops and non-regular tunings. Techniques such as *pulse width*

[31] See `https://doi.org/10.5258/SOTON/D1387`

[32] The analogue/digital converter (ADC) used to make this recording was a *Roland Quad-Capture* audio interface. Unfortunately I cannot (and do not have the required expertise to) accurately ascertain the device's preamp frequency response, harmonic distortion, intermodulation distortion and other potential attributes that *will* colour the sound and influence the captured waveform, as will the resistances of the cable and breadboard.

[33] See `https://doi.org/10.5258/SOTON/D1387`.

[34] See: `https://doi.org/10.5258/SOTON/D1387`

modulation playback of samples (see 2.2.3) are, comparatively, much 'brighter' with more high frequency components when synthesised by the microcontroller. Additionally, analogue circuitry deforms the resultant waveform, and thus timbre, away from a perfect, digital, representation. My own preference is the sound of the raw waveform; additional noise is generated in the upper frequency range, adding harmonic content where there would not be otherwise. As an aural demonstration, compare the drum samples of the software generated audio (the `mmml-generator.c` program) to the output of the microcontroller. The samples appear 'brighter' due to additional emphasis at around 9kHz and 18kHz. This companion high-frequency buzz is not unpleasant at lower volumes and produces a richer timbre. Additionally, it may be that the presence of *hypersonic* frequencies (those above 20kHz and outside the limits of direct human perception) has a positive, biological affect on textural appreciation - though the exact cause of this effect is not entirely understood [97]. Moreover, the 'minimalist' aesthetic (using as few additional components as is practicable) is both conceptually appealing and consonant with the reductionist premise of the investigation. My suggestion is that the code and structure of the 1-bit language should be communicated as directly as possible, with no obfuscation or colouration from other sonic elements. Repeated concessions may lead to increasingly more artistic 'adjustments' that do not directly originate from the 1-bit technique, for example, using a resistor ladder for adjustable volume, panning or other uses of multiple hardware channels, analogue reverberation, etc.

2.2.2 Timbre & Volume

In addition to the timbral artefacts of external circuitry, timbre can be more noticeably and predictably altered in software by adjusting a pulse wave's duty cycle. As the duty cycle decreases, narrower pulse widths are progressively quieter in the lower harmonics than wider widths, with 50% being the *perceptually* loudest width possible. Figure 7 depicts a pulse wave, decreasing in pulse width from left to right, from 50% to 0% duty cycle. The fundamental begins to decrease in intensity as the duty cycle decreases, demonstrating the reduction of the apparent 'high-pass' effect and decreased emphasis of the signal's 'bass' range. This change is continuous and is a product of the harmonic nature of the signal [98].

In fact, narrower pulses have incrementally less power *overall* to the listener; as the duty cycle approaches 0% (or 100%) the perceptual volume decreases with it. Due to the 'phase symmetry' of audio (polarity does not affect the informational, nor perceptual, properties of the waveform) this effect is not a consequence of the reduction of the signal's actual, electronic or kinetic power [35]. The reduction in volume is a product of *bandlimiting*: the limiting of a signal's spectral density to zero beyond a specified frequency. Practically, generating narrower pulse durations requires ever higher sampling rates — this is obvious, the smallest point on a temporal grid becomes its quantum (known as the 'sample') which would be the smallest duration an amplitude could occupy. The Nyquist—Shannon sampling theorem dictates that the highest frequency in a sampled waveform must be half the sampling rate (in our case, the rate of pulse wave generation) [100]. The shortest pulse *must* be two samples long; a single sample is not a change in amplitude, which, in itself, cannot be heard. We can see from Figure 8 that, at faster sampling speeds, amplitude does not decrease when the pulse width is narrow, the amplitude is a constant, 1-bit waveform. When the

[35] Calculated via *root mean square* (RMS) amplitude [99]

sampling rate is decreased however, this same waveform appears to, sympathetically, decrease in amplitude. As the sampling rate is lowered, there are two possible outcomes: either the waveform is left unchanged, but drops in frequency (lower sampling rates translate to lower cycles per second) or the frequency is retained but, in accordance with the Nyquist—Shannon sampling theorem, the signal must be bandlimited to half the sampling rate to avoid errors ('aliasing'). As we saw in Figure 5, thinner pulses are constructed from more powerful high frequency harmonics than lower and, after bandlimiting, extremely small (or extremely large) duty cycles, where the majority of the power is above the *Nyquist frequency*, are removed from the spectral density, resulting in a reduction of the waveform's overall power.

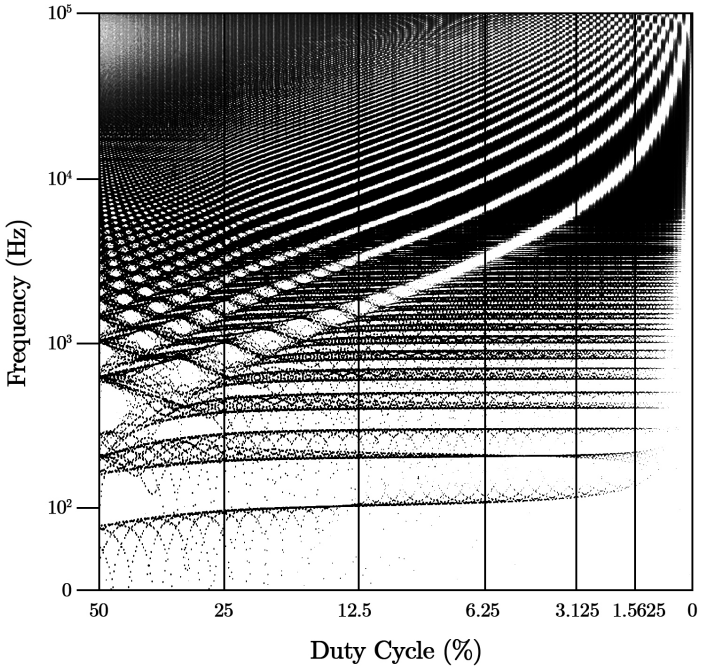


Figure 7: A spectrogram view of the *pulse width* example, where a square wave is progressively narrowed in pulse width over time. The duty cycle is decremented (non-linearly) from 50% through to 0%. Significant widths are identified by the vertical makers. The illustration here demonstrates the changes to the frequency components at duty cycles between 50% and 0% where brightness (or lack thereof) indicates the frequency's power; stronger frequencies are represented by darker lines. There is a particularly harsh cut-off threshold applied to the intensities of harmonics to highlight the aural effects of this phenomena, however all frequencies do remain present (though increasingly quieter) as the duty cycle tends to zero. The spectrogram was generated using Image-Line's *Edison* software [101]. The generated sample rate was 214000Hz and was created using the `1-bit-generator.c` program . See `pulse-width-sweep.wav` for an audio example, available at: `https://doi.org/10.5258/SOTON/D1387`.

Figure 8 has been downsampled in software, meaning it is a conceptually 'perfect' bandpass[36], which does not translate to the chaotic, noisy and unpredictable natural world; so how does the reduction in amplitude relate to perception? Filtering (in this case, *lowpass filtering* [103]) is occurring as the waveform is both propagated and sensed; upper band limited by real, physical media at every stage of transmission. Additionally, depending on the nature of the system, physical systems will add distortion and resonances (see Figure 6). 1-bit music is *acousmatic*: presented exclusively through speakers; it cannot be generated

[36]I have read, anecdotally, that software low-pass filters are often implemented imperfectly and can actually boost around the cutoff frequency [102]. I have found this to be true with FL Studio's *Fruity EQ 2*, so I cannot guarantee that, when downsampling in software, I am not inadvertently introducing errors.

naturally. As such, because the frequency response of a speaker is limited by how fast the cone can physically move, higher frequencies will not be replicated by the diaphragm. Additionally, even if the replication of higher frequencies were perfect and transmitted through an ideal, theoretical medium of infinite bandwidth, the upper limits of human perception is, unavoidably, around 20kHz [104]. Thus the appearance of amplitudinal change is caused by a conceptually perfect model translated through an imperfect environment.

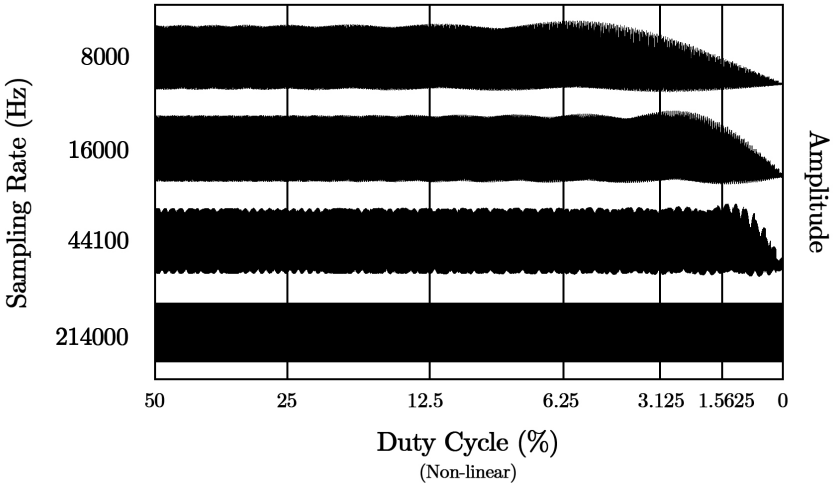


Figure 8: Four waveforms of the *pulse width* example (`pulse-width-sweep.wav`) at different sampling rates. Each has been 'downsampled' (using Image-Line's *Edison* audio editor) from the raw file. Generated at 214000Hz by the `1-bit-generator.c` program. Downsampling cuts those harmonics faster than half the sampling rate, thus reducing the power of waveforms with stronger high-frequency harmonics.

This phenomena has multiple, *musical* implications. Firstly, due to the distributions and intensities of harmonics present in different pulse widths[37], some widths are more suitable in particular traditional instrumental roles than others. For example, as human hearing is less sensitive to frequencies below (approximately) 1kHz [105] (and increasingly so as the frequency decreases), those pulse widths with stronger low-frequency partials are better suited to material that would conventionally employ bass instruments[38]. Furthermore, as it is possible to change the perceptual loudness of a voice[39] (even though only two amplitudinal states are conceptually and *abstractly* possible), the composer has the surprising ability to introduce dynamic variation into their composition. Of course, this comes with the sacrifice that timbre and volume are concomitant — in that, if one wishes to alter dynamics, one must also inextricably alter the timbre. This is less noticeable at larger widths, however it becomes discernible at widths of 5% or lower, visually demonstrated in Figure 7. If one looks

[37] Notice the striking comb filtering in Figure 7 (discussed later).

[38] Indeed, this is why thicker pulses are often found in the bass parts of 1-bit music and why one might choose certain methods of synthesis over others [106].

[39] Here the use of the word *voice* refers to a distinct instrumental line with its own sonic identity. As virtually all applications of 1-bit music are generated in software, there are no fixed 'channels' as one might expect from music drivers for *programmable sound generators* (PSGs) such as the *Nintendo Entertainment System* or *Commodore 64* [107, 108]. The entire system can therefore be considered the 'instrument', which is be unhelpful when analyzing 1-bit music. Melodic lines are generally individually discernible as consequence of common compositional practice, where voices are assigned instrumental roles reflective of traditional ensembles (for example, SATB ensemble, or wind trio). We can consider these entities 'voices': instances of sub-instruments that are given individuality by their musical application. Although no predefined channels exist, often channels are implemented in software and addressed in a similar fashion one might PSG channel. This approach has been taken in the various code examples used throughout this article.

at the center of the spectrogram, the tapering of intensity of the lowest harmonics should be evident as the width approaches 0%. It is at these extremes that a reduction in volume is perceptible. As the volume decreases, the bass harmonics perceptually recede first, sounding ever more 'reedy' or 'nasally' in texture. In contrast, those duty cycles approaching 50% sound progressively more 'rounded' and 'clarinet-like'[40] (for lack of more specific terms).

How the synthesis is *compositionally* employed plays a significant role in how apparent (and disparate) each voice's timbre-volume pairing is. Those 1-bit compositions in which thin pulse widths are exclusively utilised sound texturally homogeneous; the apparent changes in amplitude are more convincing as genuine changes in volume. Figure 2.2.2 aurally demonstrates this effect. The example is split into two arpeggios, equal in both frequency and time. Both arpeggios are composed of four dotted eighth notes, each of these subdivided into sixteenth notes. Every sixteenth note is half the duty cycle of that before it. The first sixteenth note starts at 50% (then 25% and 12.5%), the second at 6.25% (then 3.125% and 1.5625%). Despite both sets of duty cycles having identical ratios between elements (being ratios of one another to the power of two) it is only the second set, at narrower widths, that the timbre no longer appears to transform over the duration of the note, instead the apparent volume decreases. Both annotations *(a)* and *(b)* in Figure 7 show the first six partials above the fundamental for duty cycles 50%, 25% and 12.5%, and 6.25%, 3.125% and 1.5625%. The harmonics highlighted in *(a)* (those that have the greatest effect on the human perception of timbre) alter dramatically between the duty cycles, thus, over the duration of the note, these changes are heard as timbral — more a series of instrumental transformations. Comparatively, those partials presented in *(b)* remain consistent between the pulse width changes, but recede in intensity (at least in the lower harmonics), perceptually varying less in timbre, but more in volume. This behavior is a product of the distribution of harmonics in any given pulse wave; as we can see in Figure 7, the spectral image can be described by $m = \frac{1}{pw}$, where pw is the given duty cycle (expressed as a percentage) and every nmth harmonic ($n = 1 \ldots k$) will be missed.

Therefore, as the duty cycle is halved, so too is the total number of missing harmonics; the timbre becomes ever more timbrally congruous and sonically cohesive to the human listener. A thin pulse is, *practically*, a high-passed saw wave: to the human observer, nearly all integer harmonics are present, however, in the case of the pulse, with a progressively weaker bass response [110, 103]. Consequently, the 1-bit composer cannot employ a quiet voice with a strong bass component, as a reduction in volume will result in a considerable alteration of timbre and a perceptual separation of instrumentation.

The phenomenon in Figure 7 allows the 1-bit composer access to volume *enveloping*, or transforming a sound's volume over its duration [111], increasing the pulse wave's repertoire of instrumental expression. Shaping the perceived loudness via *ADSR* [112] enveloping is not only theoretically possible, but implemented in early 1-bit soundtracks such as *The Sentinel* [113] and *Chronos* [114] on the ZX Spectrum — as well as many 1-bit routines [115, 116]. Figure 2.2.2 depicts a simple implementation of ADSR enveloping on a 1-bit pulse wave, demonstrating a fast attack and decay, alongside a contrasting slower attack and decay. The 'trick' is to keep the maximum fade width to approximately 6.25%, where the initial missing

[40] I wonder if the perceived similarity between a 50% pulse wave and a clarinet is due to the similar spectral images; both clarinets and square waves produce, overwhelmingly, odd harmonics [109].

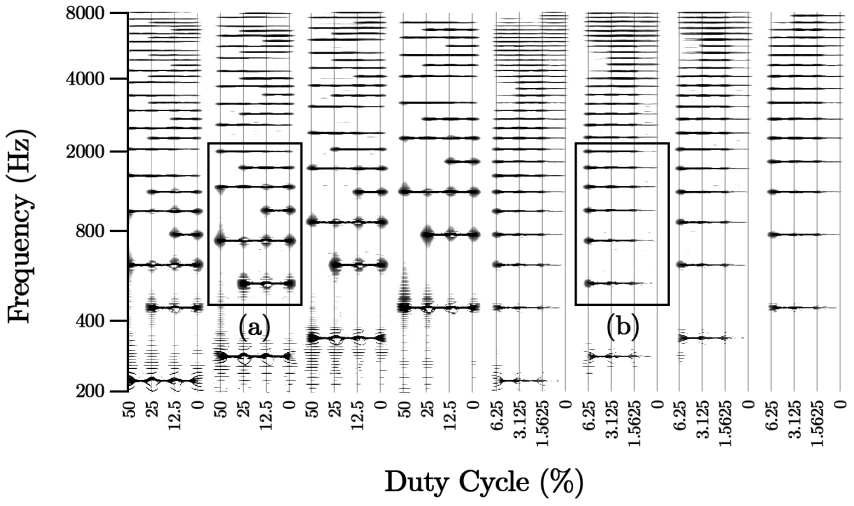


Figure 9: Spectrogram view of two, identical ascending sequences demonstrating how the timbral character of a pulse wave, in relation to the duty cycle half its width, changes as the starting pulse width is reduced. Each arpeggio consists of dotted eighth notes subdivided into four sixteenth notes of fixed frequency (one of silence), differing only in duty cycle. Each sixteenth note is half the duty cycle of the one before it. The first arpeggio starts each note at 50%, the second at 6.25%. The visualization was created using Image-Line's Edison audio editor, and the example was generated at 60000Hz by the `mmml-generator.c` program https://doi.org/10.5258/SOTON/D1387

. 6000Hz was chosen as the thinner widths afforded by higher sampling rates were not required for a minimum width of 1.5625%. See `https://doi.org/10.5258/SOTON/D1387` to listen to the original audio.

harmonic is beyond the first ten or so in the harmonic series[41], so that the fade is aurally congruous and mutates more in perceptual power than timbre. Additionally, as halving the duty cycle concomitantly doubles the change in perceptual volume, one may wish to implement a non-linear fade so that the apparent volume change is *perceptually* linear [118].

The emphasis placed on narrow duty cycles does not imply that movement between wider duty cycle is unpalatable. Envelopes that traverse wider widths modulate timbre instead, as heard towards the beginning of the example in Figure 7, between 50% and approximately 10%[42]. This effect can be employed to add further expression and individualism to discrete voices, approximating how acoustic instruments may change their spectral image over time [119]. These changing harmonics are considered *transients* and the entire envelope is recognized as an instrumental gestalt by the human listener. The `pwm-enveloping.wav`[43] and `pwm-enveloping-scale.wav`[44] audio examples are a modification of the routine used in Figure 2.2.2. This implementation ignores the previously imposed maximum duty cycle, allowing the envelope access to larger pulse widths. Each sound has its own individual characteristics; an instrumentally distinct entity. Even though generated via identical 1-bit parameters, with each sound the movement of pulse width over time forms the voice's instrumental identity. Musical examples of *pulse width modulation* (PWM), the alteration of duty cycle over time,

[41] Where variations are most noticeable; higher harmonics do not have the same influential power on the perceptual timbre [117]

[42] Of course, it should be noted that, as duty cycles beyond 50% are phase inversions of those below, sweeping to widths larger than 50% will aurally reverse the direction of modulation, appearing to 'bounce' back when reaching 50%.

[43] See here: `https://doi.org/10.5258/SOTON/D1387`.

[44] See here: `https://doi.org/10.5258/SOTON/D1387`.

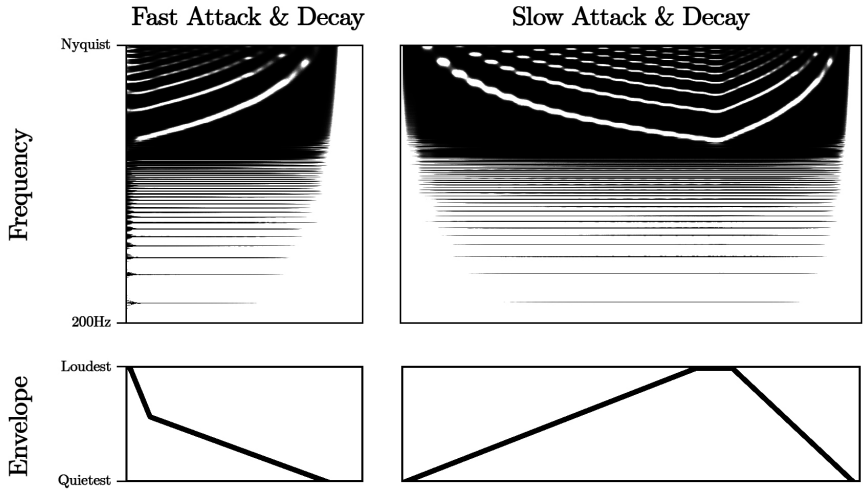


Figure 10: Two spectrograms visually demonstrating 1-bit volume enveloping. The left-hand example depicts a quick attack, starting at the maximum value, decaying to a medium, (very) brief sustain level and then moving quickly to silence. The second shows a ramp from zero to sustain, then again a decay to silence. These do not show true changes in amplitude but, instead, in pulse width. The example was generated at 214000Hz by the `1-bit-generator.c` programhttps://doi.org/10.5258/SOTON/D1387
. See `https://doi.org/10.5258/SOTON/D1387` to listen to the original audio.

for timbral effect are numerous and the approach is certainly not unique to 1-bit practice[45]. 1-bit **is** distinct however in its peculiar use of pulse width *extremity*; the utilization of very thin widths during a PWM sweep, producing a continuous blend of timbre to volume changes.

As pulse width does not affect pitch, any operation to alter duty cycle must occur within the duration of a wave cycle. Using Listing 1 as a starting point, the XOR operation can be removed and, instead, two conditions can be added to toggle the output depending on the value of the `pitch_counter` variable. In Listing 2, this value is represented by the `waveform` variable; `waveform` can be an arbitrary number (smaller than `pitch_counter`; larger values will represent duty cycles larger than 100%) or, more usefully, calculated based on a division of the frequency. Dividing `pitch_counter` by two will result in a pulse wave of 50% duty cycle, dividing by four a duty cycle of 25%, dividing by eight yields 12.5% — and so on.

2.2.3 Polyphony

On first inspection, it would seem impossible for simultaneous voices to be expressed by a 1-bit waveform. For a signal to carry two frequencies, concurrently traveling through the same medium, one would expect a superposition of one waveform on the other: a summation resulting in phase addition (or subtraction) where the interacting waveforms constructively, or destructively interfere [123, 124]. This cannot happen in a 1-bit environment; as the waveform may exist in either of two, quantized states, additional signals will fail to be represented. Figure 11 illustrates this effect with the addition of two 1-bit waveforms and the subsequent conversion back to a 1-bit composite. The product of this combination will

[45]For example, the Commodore 64's SID audio chip allowed composers access to continuous changes in pulse width, demonstrated in pieces such as Peter Clarke's famous *Ocean Loader 3* music [120] or Fred Gray's soundtrack to *Batman: The Caped Crusader*. Some 1-bit examples include Brian Marshall's somewhat crazy soundtrack to *Last Ninja 2* [121] and MISTER BEEP's brilliant *Chromospheric Flares* [122]

```
waveform = frequency / 4;

if(--pitch_counter == 0)
  pitch_counter = frequency;

else if(pitch_counter <= waveform)
  output = 1;

else if(pitch_counter >= waveform)
  output = 0;
```

Listing 2: An example of a variable pulse width generator in C pseudo-code. The `waveform` variable in example could be simply modified by changing the divisor but, as it currently exists, the example will produce a fixed duty cycle of 25%. The thinnest width possible is dependent on the size of the `pitch_counter` and `frequency` variables, with higher values yielding thinner available widths. The thinnest width will always be a `waveform` value of 1, however the proportion of 1 to the `frequency` value changes depending on the size of `frequency`. For example, 1 is 10% of 10, but 0.5% of 200.

be noisy, incoherent and will not clearly resemble either of the original frequencies. One can consider this merger equivalent to extreme distortion, or 'clipping' [125], where 0dBFS is one bit above $-\infty$dbFS, or DC zero[46].

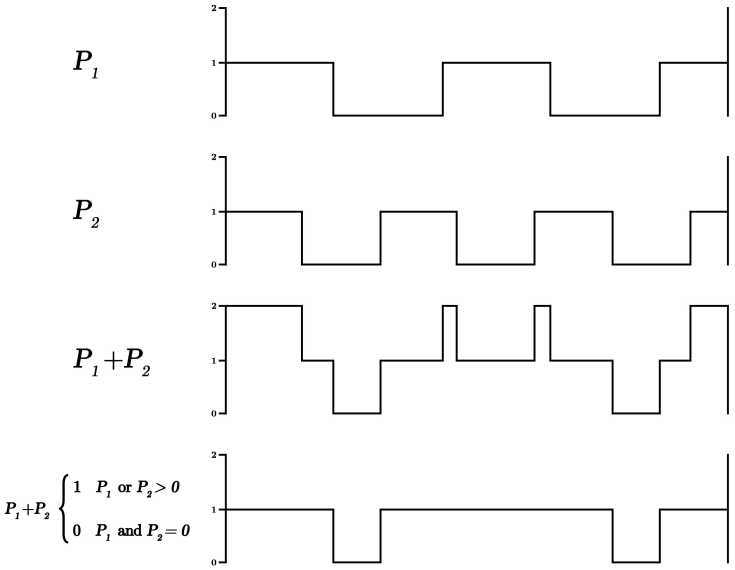


Figure 11: The summation of 1-bit signals is not the same as that of sinusoids, for example. As 1-bit waveforms are either of amplitude zero or one, they can be considered to be DC offset, where the amplitude is displaced such that the trough is always DC zero. Resultantly, there is never any wave subtraction, only addition. The reader may realize that this behaviour is equivalent to a logical OR operation — this observation will come in useful later.

Figure 11 is somewhat misleading however as it implies that the mixing behaviour at a 50% duty cycle is applicable universally. In fact, if we frame the primary issue as one of clipping, the solution to packing multiple frequencies into a binary, DC offset waveform is to reduce the rate of peak interactions and minimize distortion. This can be achieved by reducing the pulse width: as very thin pulses ($\approx$<10%) have a significantly greater space to mark

[46] It should be noted that, when employed practically, this makes little sense: 1-bit music obviously has as large a dynamic range as the system generating the signal; intermediate amplitudinal states are naturally rendered unavailable as they cannot be addressed.

ratio, the majority of waveform interactions will be trough to trough, or peak to trough, which does not affect signal identity. When two peaks eventually *do* overlap, there will still be unavoidable distortion, but the regularly of these interactions will be so infrequent as to retain the identity of the union. Therefore, we can imagine successfully merged 1-bit signals as the application of the logical OR operation on highly narrow pulse widths. This solution is called by 1-bit composer-programmer Utz as the *pin pulse method* (PPM) (or pin pulse technique) [106].

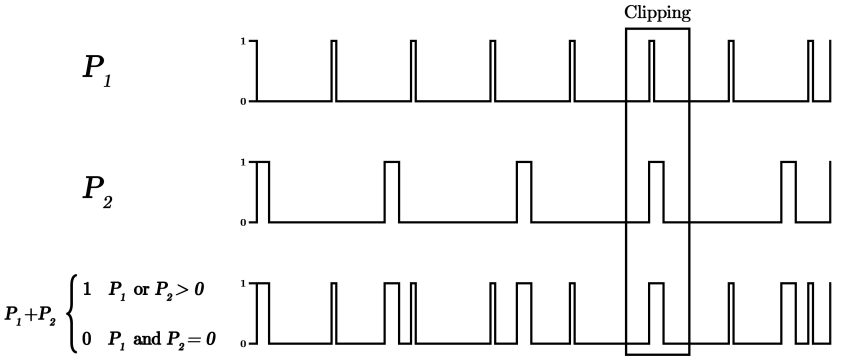


Figure 12: The summation of 1-bit signals via the pin pulse method. The identity of both P_1 and P_2 is clearly retained in the aggregate. The highlighted area indicates where clipping has occurred (the peaks have collided and exceeded the amplitudinal resolution); whilst this is unavoidable, it happens so rarely it does not affect the synthesis.

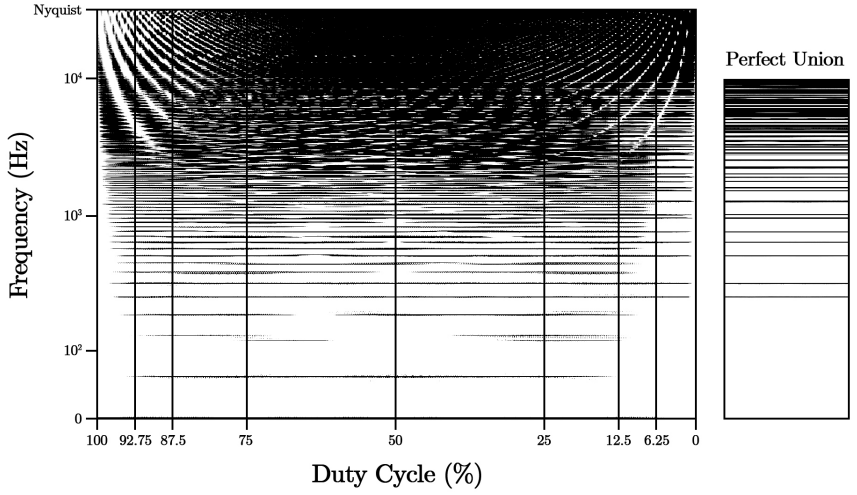


Figure 13: A spectrogram demonstration of two waveforms combined at an interval of a major third. In the example, both voices are modulated from 100% through to 0% duty cycle. A perfect union (of saw waves — a useful reference as they contain all the integer harmonics) is shown for reference; note how the final combination of pulses at (≈<6.25%) is almost identical to a true summation. The visualization was created using Image-Line's Edison audio editor and the example was generated at 64100Hz by the `1-bit-generator.c` program. See `https://doi.org/10.5258/SOTON/D1387` for an audio example.

Figure 13 is a visual representation of two 1-bit waveforms summed in this way, at an interval of a major third. Towards the end of the modulation (between 12.5% and 6.25%), notice how the harmonics become less 'noisy', more stable and accordant with the expected spectral image (the image that would be seen if the combination were at a higher amplitudinal resolution, shown to the right for reference). Surprisingly, the final combination of harmonics

are preserved across the entire image (see the two, bottommost lines at duty cycle 6.25%), meaning that the combination is largely present across all pulse widths, just with varying levels of additional noise. This 'gritty' texture, whilst not a perfect representation of the ideal signal, can be employed to musical effect and, if the composer does not find a certain level of distortion unpalatable, can still communicate coherent polyphony[47]. This noise can be used to 'thicken' a 1-bit piece, busying the soundscape to apply texture. Figure 14 utilizes this gradual distortion to pleasing effect; sweeping a $\mathrm{B}(+15¢)^{\mathrm{add9}}$ chord[48] from duty cycles 0% to 100% to produce a sound not dissimilar to incrementally increasing the input gain on an overdrive guitar pedal [126]. Figure 14 highlights another benefit of this technique: the pin pulse method is so effective that numerous frequencies can squeezed into the same signal; simply apply the logical OR operation to any new waveform onto the existing composite. As a general rule, the more pulse 'channels', the thinner the pulse widths must be before distortion; with every additional signal, the chance of a peak–to–peak interaction is ever more likely. Note, this technique does necessitate high sample rates and, the higher the sample rate, the thinner the pulses can be. This means that the chances of a peak interaction event is increasingly reduced. Of course, there is a practical upper limit on this as, discussed previously, as pulse widths decrease, the perceptual volume also decreases.

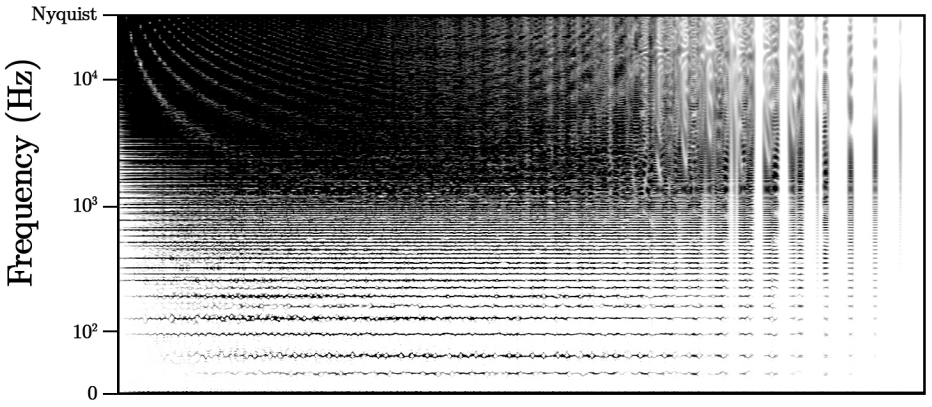


Figure 14: A spectrogram view of five 1-bit waveforms combined using PPM and widened from 0% to 100% duty cycle. The chord gradually becomes increasingly distorted as it decays. Interestingly, the example has some noticeable periodic interference, known as *beating*, intermittently boosting the root of the chord (B+15¢). This beating is more severe towards the end of the spectrogram, where there is oscillation between 'silence' (the voices at this point are at 100% duty cycle) and sound. The visualization was created using Image-Line's Edison audio editor, and the example was generated at 64100Hz by the `1-bit-generator.c` program. See `https://doi.org/10.5258/SOTON/D1387` for an audio example.

To implement PPM in software, for each sample, one must execute a bitwise OR operation (the '|' character in C) on all software outputs, then update the hardware output with the resultant value. Rather than updating the hardware output directly, in order to combine

[47] It should be noted that, if the waveform was inverted so that 0 was 1 and 1 was 0, Figure 13 would reverse so that it became progressively *more* distorted. I assume in this article that 0 is off and 1 is on, following general convention. There is no 'chirality' so to speak, the speaker may oscillate between *any* two states and the theory would still hold. In the event of reversing the signal logic level, Figure 13 would sound most consonant towards 100% and distorted approaching 0%.

[48] Because the relationship between notes is, musically, more important than their actual frequencies, I have paid little attention to pitch standards. The reader may notice that all examples/pieces will be in different keys and tunings. To guarantee a piece is at the 440Hz standard requires either clocking the routine against a timer, or carefully calculating frequencies — which is certainly practicable, however I personally have not put too much weight on ensuring that it is correct.

multiple channels, a virtual output must be created for each voice. These virtual outputs are never directly sonified, but are combined; it the resultant which is applied to the hardware output, creating the final waveform. Listing 3 demonstrates how this might be implemented in C. Although two channels have been used in this example, there is, theoretically, no limit to the amount of software channels that can be combined using PPM; one can continue append additional outputs with the bitwise OR operation. With each successive channel, one must employ progressively thinner widths to mitigate the increased probability of pulse collisions.

```
// process voice #1
if(--pitch_counter_1 == 0)
  pitch_counter_1 = frequency_1;

else if(pitch_counter_1 <= waveform_1)
  software_output_1 = 1;

else if(pitch_counter_1 >= waveform_1)
  software_output_1 = 0;

// process voice #2
if(--pitch_counter_2 == 0)
  pitch_counter_2 = frequency_2;

else if(pitch_counter_2 <= waveform_2)
  software_output_2 = 1;

else if(pitch_counter_2 >= waveform_2)
  software_output_2 = 0;

// combine software outputs
hardware_output = software_output_1 | software_output_2;
```

Listing 3: A demonstration of PPM mixing in C pseudo-code. Although all channels have been addressed individually in this example, the companion `1-bit-generator.c` program (`https://doi.org/10.5258/SOTON/D1387`) employs a `for` loop to iterate through software channels and update outputs. This makes it easier to add or remove additional channels without requiring additional code and variables. When transcribing this example for other platforms (for example, ZX Spectrum machine code), one will probably find the paradigm used above (explicit declaration of each channel's processing code) more efficacious.

Whilst PPM is an effective method of crowding numerous frequencies into a single, 1-bit waveform, the strict requirements of the technique sacrifices timbral variation for polyphony. As shown previously, square waves — perceptually the loudest width possible with the 'richest' bass component — cannot be employed without heavy distortion; a square wave will interfere for 50% of its signal high duration. When juxtaposed with analogue combinations of square waves, music made via PPM will sound contrastingly thinner, quieter and lacking timbral 'substance'. One potential, beneficial consequence of this concession is enforced aesthetic cohesion. The technique's demand for narrow duty cycles means that the dilemma presented in Figure 2.2.2 (PWM between larger widths is perceived less as a change in volume, but instead a change in timbre) is circumvented by the process; the PPM 1-bit routine sounds like a single, cohesive instrument.

The second method of achieving polyphony does not suffer from a narrow timbral palette; this method is known to the 1-bit community as the pulse interleaving method (PIM) [106]. One can imagine the implementation of this technique as, essentially, simultaneity by rapid

arpeggiation. PIM operates on a fascinating premise: rather than mixing signals logically to fit within amplitudinal limitation (as with PPM), the technique switches, or arpeggiates, between voices at very high frequencies, so that only one, monophonic waveform is expressed at any moment in time. The rapidity of oscillation between channels needed to achieve convincing polyphony is not particularly fast, however a problem arises as a result of quickly moving between two states: toggling a waveform from high to low (or vice versa) creates a click, or pulse. As with any pulse wave, because alternation between voices must happen at constant periodicity, this click produces a pulse train and, consequently, an audible, pitched sonority at the rate of interchange. To disguise the additional, audible "parasite tone", the software mixing must happen either faster than human auditory perception, or higher than the frequency response of the medium replicating the signal. This familiar requirement suggests something intriguing: just as thin pulse widths, when bandlimited, become true changes in amplitude, as long as frequencies beyond the parasite tone are removed, the union can be considered perfect: true polyphony.

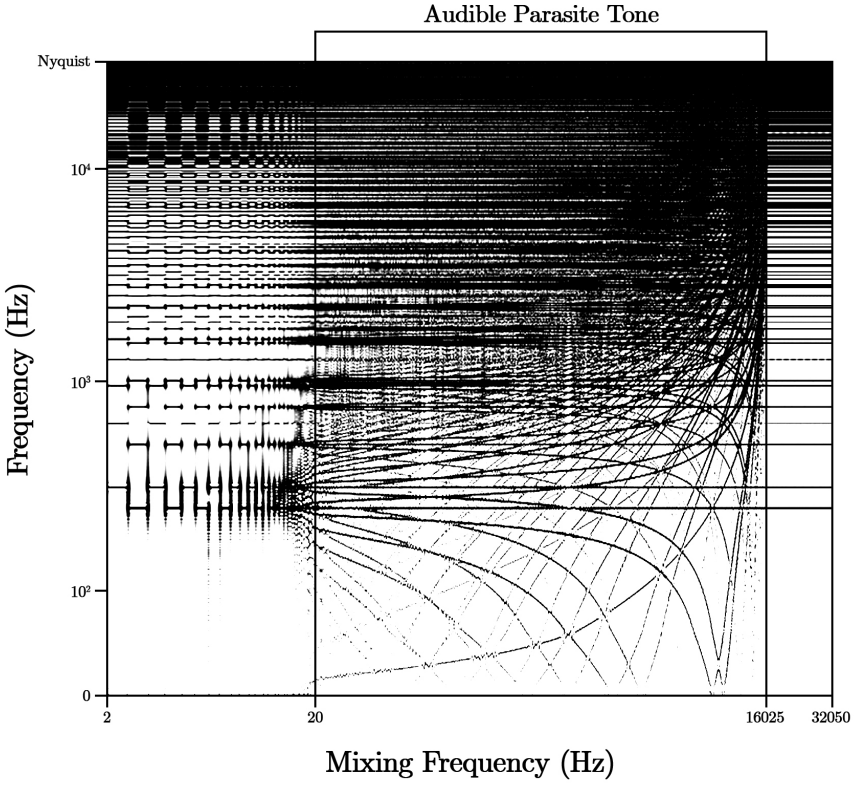


Figure 15: A spectrogram view of two 1-bit waveforms gradually combined using the pulse interleaving method. Pulses are mixed from 2Hz through to 32050Hz (Nyquist) to demonstrate how the parasite tone is generated then pushed upwards, beyond audible frequencies. The mixing speed does not *have* to happen at Nyquist, but it often will do so through implementation. Simply, with 1-bit music you will generally want to 'spit out' samples as fast as practicable; if the switching is occurring at the maximum frequency possible, there will be a change of output at every sample and, consequently, a waveform generated at the Nyquist frequency. The visualization was created using Image-Line's Edison audio editor, and the example was generated at 64100Hz by the `1-bit-generator.c` program https://doi.org/10.5258/SOTON/D1387 . See `https://doi.org/10.5258/SOTON/D1387` for an audio example of the unison and `https://doi.org/10.5258/SOTON/D1387` for the isolated parasite tone (the waveform generated by the mixing).

A bandlimited PIM waveform has a higher amplitudinal resolution than its carrier wave

but, rather than increasing volume when waveforms are merged, additional signals are progressively, *individually* quieter. To explore this phenomenon, a similar thought experiment can be applied to that mentioned previously when examining variability of volume. The effect of bandlimiting, or more accurately in this scenario, low-passing, can be explained by imagining how a speaker cone might move when a PIM signal is applied to it. A hypothetical diaphragm with a maximum response time (moving from stationary to fully extended[49]) of f–Hz, is unable to react instantaneously to changes in voltage faster than this response time. If a signal's frequency exceeds f (let us assume the transmitting function is square) the diaphragm, after receiving a positive voltage, will move only a portion of its total distance before the signal falls to zero and the cone sympathetically returns to rest. Therefore, if we send two, concurrent 1-bit square waves to the speaker and oscillate between them faster than f, the diaphragm is unable to complete a full extension. There is, to the speaker, no parasite tone, instead we observe the following behaviour, shown in Table 1.

P_1	P_2	Output
0	0	0
0	1	0.5
1	0	0.5
1	1	1

Table 1: A table showing the resultant, comparative perceived loudness of signals mixed via PIM. P_1 and P_2 are 1-bit waveforms of arbitrary pulse width (though very thin duty cycles *will* lower the perceptual volume of that voice, but not affect the ratio between the others) and the output values represents the mixed volume level in response to different input states.

The signals P_1 and P_2 are digital and can only exist in two states, 0 or 1. The output position of the diaphragm is assigned 0 at rest and 1 at maximum extension. The diaphragm will always attempt to act concordantly with the signal, however it can be 'tricked' into generating a third state. This third position is an intermediate, caused by moving the speaker cone faster than it can respond, leaving the diaphragm hovering at the average of the two voltages. We can see from Figure 16 that, when bandlimited in software, the behaviour is identical to that of the thought experiment; a $1\frac{1}{2}$-bit signal emerges from filtering[50]. The benefit of performing this in software is that a frequency of arpeggiation can be arbitrarily assigned as long as a low–pass filter at corresponding frequency is applied, completing the union. It is important to remember that the thought experiment does not explain the most common reason of bandlimiting in the real world: as many speakers will replicate signals above the human, biological maximum, the frequency response of the ear is the only variable that can must be mixed beyond to reliably ensure the parasite tone is removed.

The pulse interleaving method is not limited to two signals alone: depending on the frequency (the rate of oscillation between *all* virtual 1-bit signals must exceed the aforementioned f) more channels can be added with the trade-off that, with each channel, each individual signal's volume is ultimately quieter (and the computational demand increases, making it harder to reach f). The total volume of each subsequent channel is a subdivision of the maximum power equating to $v = \frac{1}{c}$, where v is the volume compared to the maximum and c is the total number of channels. The ratio between elements can be altered, however this will skew the relative loudness of outputs, as mixing priority will be assigned

[49] In either direction. 1-bit signals are conceptualized as DC offset, so that oscillations occur from amplitude 0 to 1, where 1 is the maximum power possible in the system.

[50] $1\frac{1}{2}$-bits representing three possible states.

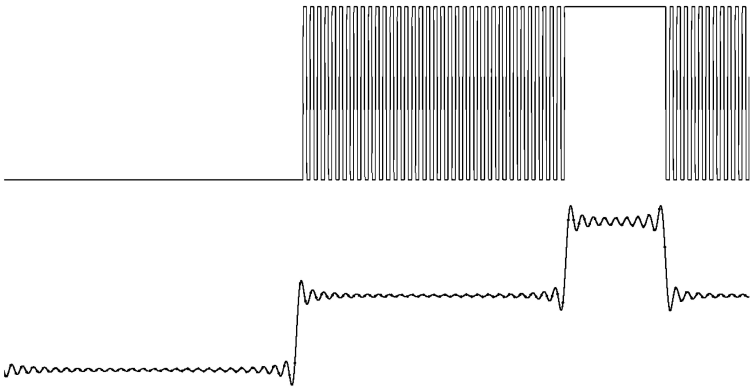


Figure 16: A (software) oscilloscope view of a PIM waveform generated at 64100Hz by the `1-bit-generator.c` program (`https://doi.org/10.5258/SOTON/D1387`) and, below, the same waveform downsampled (applied lowpass and resampled) to 44100Hz.

unequally. To achieve more complex ratios (not a simple $\frac{1}{n}$) requires higher sampling rates where, rather than cycling equally between unique outputs, a singular channel is expressed for more samples than another[51].

For a complete description of PIM, we must explore the second, significantly faster, usage of PWM. Using a pulse width modulated pulse train, it is possible to generate continuous amplitudinal variation using a digital signal alone[52]. Pulse width modulation describes the method by which a succession of pulses are delivered[53] to regulate the voltage of a signal. When this signal is applied to a speaker, or similar output device, it acts as if it were receiving an analogue signal[54]. Conceptually, it is not *too far* removed from the previous 1-bit theory covered here, but requires faster rates of update to achieve: the operable bandwidth is approximately ten times smaller than the switching frequency [127]. The behavioral change from a digital to analogue signal can be described by the signal's root mean square amplitude [99]. The root mean square is the average power of the waveform, proportional to the area under the waveform's curve when transformed in this way. Consequently, the emergent analogue signal is a product of the average value of the waveform, when modulated on a fixed frequency, and appears, to the system receiving the signal (this medium would be considered the 'integrating' mechanism), to be a true analogue signal (if that signal responds the changes slower than the frequency of the carrier wave). The pulse interleaving method can therefore be considered as a low resolution PWM. This is *much* more computationally friendly to implement in software however, as the bit depth need only be the desired number of voices. PIM employs each 'bit' of PWM resolution to convey an individual 1-bit channel, capable of independent duty cycles.

There are two main approaches one might take to implement PIM in software. The first of these methods is to simply update the hardware output directly (in a similar fashion to

[51] This can be a nuisance if clock cycles between updating outputs are unequal, for example the last channel outputted in a loop might remain active for the remainder of the code making the perceptual volume of this channel louder than the others. This can be avoided (or embraced) by careful CPU cycle management.

[52] Or pulse density modulation (PDM) but, due to the relative similarity between the two approaches, this investigation does not explore this method.

[53] At certain frequencies.

[54] Do note, PWM is used for many applications other than audio (controlling motors, LED brightness, telecommunications, etc.).

listings 1 and 2) but allow the mixing to be controlled by the program *control flow*. Listing 4 demonstrates how this might be achieved; the output is interleaved by the structure of the program: after the first channel is calculated and its value written to the output, the second is subsequently calculated and the current output is replaced with this new value. The program then loops around to replace the second channel's value with an updated value from the first — and so on. The caveat to this approach is that the amount of time taken to process the code between each output command must be identical to attain equivalent time spent expressing each channel's output value. If this is not achieved, the perceptual volume of the channels will be unbalanced[55]. The second method is to manually control the interleaving by using an incrementing variable to switch between software outputs, shown in Listing 5. With this technique, the outputs are balanced by the fact that, each time the code reaches an instruction to change the output, the channel to be expressed is switched.

```
// process voice #1
if(--pitch_counter_1 == 0)
  pitch_counter_1 = frequency_1;

else if(pitch_counter_1 <= waveform_1)
  output = 1;

else if(pitch_counter_1 >= waveform_1)
  output = 0;

// process voice #2
if(--pitch_counter_2 == 0)
  pitch_counter_2 = frequency_2;

else if(pitch_counter_2 <= waveform_2)
  output = 1;

else if(pitch_counter_2 >= waveform_2)
  output = 0;
```

Listing 4: A demonstration of PIM mixing in C pseudo-code, interleaved via the program control flow. The interleaving occurs at the speed the program can process each command thus may cause inconsistent comparative loudness between channels.

It should be added that I do not really consider high-rate PWM *alone* an *aesthetic* 1-bit solution; I feel that it is distinct in that the goal of PWM as a technique is usually to be *imperceptible*, existing only as the signal it is attempting to recreate. Although there are examples of PWM samplers implemented in 1-bit audio routines (there are examples of contemporary 1-bit routines which rely solely on this method, such as Utz's *stringks* engine [128]) these are always (technically) *poor* implementations[56], in that they produce artefacts — the process is audible to the listener. Embracing a medium's distinguishing imperfections is part of the ethos of wider chipmusic practice, where platform authenticity is contextually important to the artefact [52]. This may inform how the composer will wish to design their software routine. The seeming process duality of PPM versus PIM is reliant on two assumptions: firstly that the composer wishes to use polyphony at all (for example,

[55] We can cheat in the companion `1-bit-generator.c` program. In this case, as samples are written to a buffer (to eventually build a wave file), the time taken to execute code is meaningless as long as each contiguous sample in the buffer represents a different channel's amplitudinal state.

[56] Although, personally, I find them far from timbrally poor — they are both musically interesting and technically impressive!

```
// process voice #1
if(--pitch_counter_1 == 0)
  pitch_counter_1 = frequency_1;

else if(pitch_counter_1 <= waveform_1)
  software_output_1 = 1;

else if(pitch_counter_1 >= waveform_1)
  software_output_1 = 0;

// process voice #2
if(--pitch_counter_2 == 0)
  pitch_counter_2 = frequency_2;

else if(pitch_counter_2 <= waveform_2)
  software_output_2 = 1;

else if(pitch_counter_2 >= waveform_2)
  software_output_2 = 0;

// switch software outputs
if(current_output == 0){
  hardware_output = software_output_1;
  ++current_output;
}
else if(current_output == 1){
  hardware_output = software_output_2;
  current_output = 0;
}
```

Listing 5: A demonstration of PIM mixing in C pseudo-code, interleaved via a switching variable. One might also use software outputs and place `hardware_output` update commands throughout the code, ensuring that the timing is at equal between each update — it is certainly a visually messy solution, but I have found it to work rather well on embedded systems! The switching variable ensures that each output is expressed for an equal amount of time, keeping loudness uniform across channels.

Rockman on the ZX Spectrum [129] and the DOS version of *The Secret Of Monkey Island* [130], both of which employ monophonic arpeggiation between 'simultaneous' voices to imply polyphony), secondly that the composer is using embedded, or legacy, architectures (such as the ZX Spectrum, or early IBM PC). The concessions and limitations described above are considerations one must make when there is a technical restriction placed upon the music. If the composer wishes to merely evoke 1-bit instrumental aesthetics, then one might opt for the most idiosyncratic approach — or that which is most recognizably 1-bit. Brian Eno's observation that the signature imperfections of a medium become its defining characteristics, is certainly apt in this situation [50]. I would personally argue that *the* instrumental 1-bit idiolect, irrespective of chosen platform, is PPM. Although those 1-bit routines which employ PIM are often imperfect in practice, when executed by a system capable of perfectly executing commands at the correct frequency, PIM is indistinguishable from true polyphony.

There is certainly still room for innovation; contemporary 1-bit practice has had a comparatively brief period time to develop and explore technique compared to other instrumental models, such as the orchestra. 1-bit sonics were only commercially relevant for perhaps a decade before relegation to obsolescence and relative obscurity. It seems a shame to abandon what is a unique aesthetic environment with musically interesting instrumental capabilities, presumably in the pursuit of increased realism. This said, contemporary routines

are still being developed and can be far more sophisticated than legacy implementations; some boasting 15 concurrent voices and others sample playback [131].

3 Implementation

3.1 The 1-Bit Sound Routine

The 1-bit palette is capable of communicating many of the chipmusic techniques typical of the *Nintendo Entertainment System*, *Game Boy* and *Commodore 64*[57] however, unlike these systems, the limitations are not dictated by audio hardware, but by the *software* implementation. 1-bit music is typically generated by the CPU alone, often requiring copious amounts of calculation time. This is both the medium's greatest weakness[58] and greatest strength, as the synthesis can be determined in software alone and dictated entirely by the composer-programmer.

I use the terminology "composer-programmer" to describe the individual who not only creates music but also its software encapsulation, the *sound routine*[59]. A sound routine is a program, function, or portion of executable code that generates audio, generally with a focus on synthesising music. The term 'routine' is used synonymously with 'software' or 'program' due to demoscene and chipmusic nomenclature [106, 137, 61][60]. This expression is derived from 'subroutine', probably owing to the fact that many of these programs were created as smaller subroutines within a larger program [139]. This is an immensely exciting medium for the musician; the key input of the composer-programmer in this framework is their creative expression when designing routines for musical playback. The choices made in the design of the routine dictate which musical possibilities are available in subsequent compositions. There are multiple 1-bit routines for the ZX Spectrum that implement identical concepts and synthesis techniques, yet each of these routines develop and expand on different facets, making concessions on one feature for increased focus on another. For example, Tim Follin's early *3ch routine* [135] forgoes accurate pitch replication and quantity of channels for smaller filesizes, whereas *Jan Deak's ZX-7 engine* implements eight software channels with more accurate tunings, but makes concessions on RAM buffer space [140]. This constant negotiation between driver features is a limitation of the trade-off between memory and performance.

Completeness of musical representation is valued in a notation system. The more musically flexible a routine/system becomes, the less succinct it becomes. What is left out of the program is what gives the program its character. This section concerns itself with the implementation of Section 2 and should provide the reader with a guide to the fundamentals of writing 1-bit software routines, the rationale behind the primary routine utilised in the project and a technical explanation of how the research's music is both sequenced and synthesised.

[57] For an audio example, I direct the reader to the musical language of the following ZX Spectrum pieces: raphaelgoulart's *surprisingly NOT four twenty* [132] and Brink's *M'Lady* [133], both of which follow the classic programmable sound generator (PSG) paradigm. In short, this is characterized by instrumental figures such as super-fast arpeggios and the treatment of single oscillators as individual instruments [8].

[58] On slower systems 1-bit routines may leave too little processing time for other tasks, such as, in the case of the ZX Spectrum, gameplay.

[59] There are many examples of this individual in 1-bit, and chipmusic, history. To name a just few: David Warhol [134], Tim Follin [135] and Dave Wise [136].

[60] Alternatively, *driver* is sometimes used — associated with music generation routines; commonly referring to event sequencing programs written to interface with outboard PSGs [138, 134].

To better depict the relationship between the sound routine, the platform and the music data, one might draw parallels with other, more conventional, instrumental practices. On many legacy systems, one can consider the sound routine to be the performer. Computers, such as the Commodore 64, have dedicated audio generation hardware that receives instructions from the CPU [108] and, subsequently, creates sound. Just as a guitar cannot spontaneously pluck its own strings, this outboard hardware requires input to function. Similarly to the guitar, the audio hardware has a potential dictated by its performer. One can design a sound routine with nuanced and expansive techniques, or one can design something relatively simple; the instrument is only as capable as its instrumentalist. The data read by the sound routine would be analogous to the performer's score[61] and any generative functions within the routine[62] might be considered extemporaneous material. Where the audio hardware and CPU sit within the same unit (as opposed to a computer controlling an outboard synthesiser, where the computer holds the music data and routine and the synthesiser provides the audio generation), an analogy might be of the entire system as a 'venue'. The 1-bit routines written for platforms such as the ZX Spectrum [131], and the microcontrollers of this project, are interesting in that the sound routines often form both performer and instrument. The routine has a function to generate audio as well as read music data to control this instrument. This type of routine is perhaps more similar to a vocalist, where the performer *is* their own instrument.

Before we consider potential software implementations (and ultimately compositional strategies), it is useful to examine exactly how 1-bit music (and indeed all music) is ultimately constructed. Music can be imagined as a series of repetitions at different speeds: a chord sequence may repeat every twenty seconds, an ostinato every two seconds, and a beat every half-second. This extends 'downwards' (or faster) to the repetition of single wave cycles. For example, the note A requires repetitions at four hundred and forty times a second. This perspective is particularly useful when analysing software routines as it dictates at what speeds different events must occur. For example, a system to repeat waveform events must be created plus a further, additional system for note events. Due to the nature of a 1-bit signal, the arrangement of amplitudinal events over time becomes a principle resource for introducing sonic nuance in 1-bit music. The extent of this variance is dependent on the temporal resolution, or *sample rate*, with lower sampling rates requiring radically different treatment to higher rates. A deconstruction of temporal structures into discrete, *time scales* of music, clearly demonstrates the function of this phenomenon.

In *microsound* [141], Curtis Roads divides the entire gamut of musical temporal spectra into nine distinct ranges. Whilst comprehensive, in respect to 1-bit music, only the *macro, meso, Sound-Object, micro and sample ranges* are required:

- *Macro*: Generally measured in minutes and hours, the macro scale is the duration of musical form. Unlike the temporal phylum beneath it, the macro is perceived *intellectually* as an aggregate of meso structures, building familiar musical architectures such as *sonata, ternary and rondo* forms. Macro structures are understood retrospectively; whilst they may be obviously delineated by starting and ending cues, the interior of the object is not usually recalled in a strict, linear organisation.

[61] For this project, this would be the compiled μMML bytecode.
[62] For example, those techniques described in Section 4.4.2.

- *Meso*: The meso scale is the domain of melodic and harmonic interaction, known to Wishart as the *sequence* [142]. It is characterised by phrasings and groupings of Sound-Objects. These are the constituent parts of melodies, chord progressions, metric patterns and harmonic movements. Events at this level last between several seconds to a few minutes and define a piece's musical syntax: the grammar that allows coherency at the macro level.

- *Sound-Object*: The length of an individual note, the "elementary unit" of composition. These are the discrete 'words', lasting a few tenths of a second to several seconds, that make up the musical lexicon at the meso domain. A Sound-Object is the smallest sonic entity whose order can be clearly differentiated. Repetition of events at the extreme edge of this range perceptually blur into a continuous tone. As with the higher order categories, the Sound-Object is constructed of a series of amplitudinal oscillations at the order below it.

- *Micro*: This class spans the boundaries of the audio frequency range, responsible for all sonic events recognised as a pitch. It is the domain at which a single pulse of a rectangle wave exists, or the complete wave shape of a sine wave. Events at the micro speed do not have to be a repetition of regular tones, like the 1-bit square wave. As Roads remarks, events can be unpitched, percussive or "bursts of infrasonic flutterings". Speeds here range from to hundreds of milliseconds through to hundreds of *microseconds* in length. Streams of events become 'visceral' tones with texture rather than rhythm. This is the scale at which PPM is mixed.

- *Sample*: This is temporal grid to which the smallest events in digital music are quantised. This sub-micro time scale is measured in microseconds and is defined as the region where separate samples (an amplitudinal position at a fixed point in time) are expressed. In computational systems, this is the speed at which each point on the waveform's Y axis requires updating to generate an audible signal. For individual events, whilst still perceptible (a rapid change in amplitude will be perceived as a click), temporality becomes confused. The sample scale is the most fundamental level of 1-bit music (and any digital musical) and approaches the maximum obtainable speed for many legacy computers and contemporary microcontroller architectures.

  The range of possible sample rates dictate the type of synthesis possible: grids with lower resolutions form the basis of *aesthetic* 1-bit music, the subcategory of chipmusic, whereas higher rates allow for 1-bit DSP techniques such as delta sigma modulation and pulse density modulation [143]. As mentioned in Section 2.2.3, whilst this is not entirely beyond the scope of 1-bit music as sampling rates for PDM can drop within the micro range (and the processors/microcontrollers usually generating this music have the ability to push into the sample domain, even under load). Low sample rates when playing back sampled audio results in low-passed, 'muddy' waveforms however so, for 1-bit music on less capable platforms, a slower, oscillator-driven approach is *usually* more palatable[63].

These categories can be subdivided again using Åkesson's model of 'stacked' frequency structures [62]. Whilst this model is implicit in Roads' time scales, Roads does not

[63] Though it can sound pretty gnarly and works very well for percussion. Used in `mmml.c`, see Section 3.2

specifically make reference to the sonic repetitions at these scales. Åkesson divides music into four ranges of frequency perception, or *frequency domains*: *Structural, Rhythmic, Effect and Pitch.* These are demarcated by their repetitions per second, rather than purely their durations. Each structure can be seen as an *emergent* property of the supporting, higher frequency structure below it. Åkesson's model is particularly useful (and appropriate to the investigation) as it has been conceptualised to explain the rationale behind programmed music in the chipmusic paradigm.

- Cyclic events at the *Structural* level take place at the meso scale, from roughly 0.1Hz to 0.001Hz, corresponding with repeating phrases of musical materials.
- The *Rhythmic* domain occurs at 10Hz to 0.1Hz; event repetitions at this speed are perceived as rhythms and sit in Roads' Sound-Object bracket.
- The *Effect* domain is a new event category, sitting between the Sound-Object and micro domain. Repetitions here, at around 100Hz to 10Hz, generate many popular chipmusic effects such as super-fast arpeggiation and PWM 'sweeps'. Some analogues in wider music practice are the speed of a low frequency oscillator (LFO) or a textural modulation. This periodicity is important in Åkesson's model as it was historically the maximum speed some early computational systems, relying on hardware function generators, could manipulate their waveforms.
- Finally, the *Pitch* domain is equivalent to the micro scale. Repetitions here occur at 20kHz to 100Hz and result in 'pitched' sounds.[64]

In essence, the 1-bit sound routine can be fundamentally seen as a pragmatic application of this framework. To demonstrate this, the anatomy of a playback routine is generally split into four distinct sections and might be constructed as follows:

- Navigating blocks of musical data at the Structural Domain.
- Updating pitches at the Rhythmic Domain.
- Timbral and parametric manipulation (vibrato, sweeping pitch) at the effect range.
- Function generator at the frequency domain (mixing waveforms via PPM is here, one of the benefits over PIM).
- Mixing waveforms via PIM, or implementing PDM, at the sample domain.

Whether pitches and structures are read from sequenced data, or defined generatively, all routines will perform operations somewhere within these bands. 1-Bit Symphony composer Tristan Perich gives a concise description of this process when interviewed about his programming methodology [144]:

> *First you have the code that generates a tone. Then you set up another piece of code that can change that tone every once in a while.*

[64] One can hear an example of a pulse wave modulated through all these time domains here: `https://doi.org/10.5258/SOTON/D1387`

Tautologically, the most fundamental (and integral) element in an audio routine is the audio. On platforms that rely on 1-bit synthesis (for example, the ZX Spectrum), this is produced by toggling an output pin to apply pulses to a diaphragm [7], as outlined in Section 2.2. To produce a tone, *all* beeper routines will have to perform the following in software:

- Set a desired frequency.
- Set a timer.
- Check to see if the timer is equal to a proportion of the period equal to the desired duty cycle.
- Change a pin's output state (update the waveform).

Obviously, the above can be manipulated and interpreted in a variety of ways[65], but all remain semantically congruent. As periodicity and regularity is required to create frequency [83] there *must* exist an entity to regulate consistent intervals between amplitudinal events, otherwise, as previously mentioned, irregular intervals will result in unpitched sonorities[66]. This counter can be a custom implementation in software, or compared against a hardware timer ('clocked' [145])[67]. The `square.c` program (Listing 6) generates a single square wave, demonstrating perhaps the most elementary 1-bit tone generator and its composition:

```
#include <avr/io.h>
#define DELAY 12

int main(void)
{
  DDRB = 0b00000001; // Set PB0 to output

  uint8_t   pitch_counter  = 0,
            frequency      = 255;

  while(1)
  {

    if(pitch_counter-- == 0)
    {
      pitch_counter = frequency;
      PORTB ^= 1;
    }

    for(uint8_t i = 0; i < DELAY; i++)
      asm("nop");
  }
}
```

Listing 6: Simple AVR C square wave tone generator (`square.c`: `https://doi.org/10.5258/SOTON/D1387`).

Here we can see a scripted version of the aforementioned list of basic operations required for tone generation. The desired frequency can be set with the `frequency` variable. The timer variable `pitch_counter` is regulated somewhat with the delay function, but decrements

[65] It gets a bit weird, see Section 4.4...

[66] Which can be exploited to musical effect; unpitched sonorities work brilliantly when employed as percussive instruments.

[67] As seen in Alex 'Shiru' Semenov's *Octode* port to the Arduino platform where, from lines 108 - 118 (sample rate is dictated by timer interrupts [146]) `OCR2A` sets the frequency.

continuously within the main loop of the program[68], checking if zero at each evocation. When this is valid, it is set to the value of `frequency` to be compared against 0 indefinitely. This provides regularity and, when the `if` condition is true, I/O port logic level (beget voltage, beget amplitude) can be alternated to create frequency. As long as `frequency` is not varied at the sampling rate, the equal intervals between pin XOR operations (line 42) produces a pulse wave of equal high and low durations: a square wave.

1-bit music itself provides a striking platform to demonstrate the fundamental operations of a 1-bit routine. When viewed as amplitudinal events of different temporal successions, 1-bit music can be considered somewhat fractal in nature. For example, individual notes become slow oscillations modulating repetitions of wave cycles at higher frequencies. Figure 17 is a visual representation of this interpretation built using Åkesson's model of musical temporal ranges. The topmost waveform one would experience as a percussive pattern — a gate that, when on, allows waveform activity on or off; the second waveform is perceived as a timbral effect, aurally akin to 'growling' on a wind instrument; and the third waveform will be heard as a tone (essentially, ring modulation).

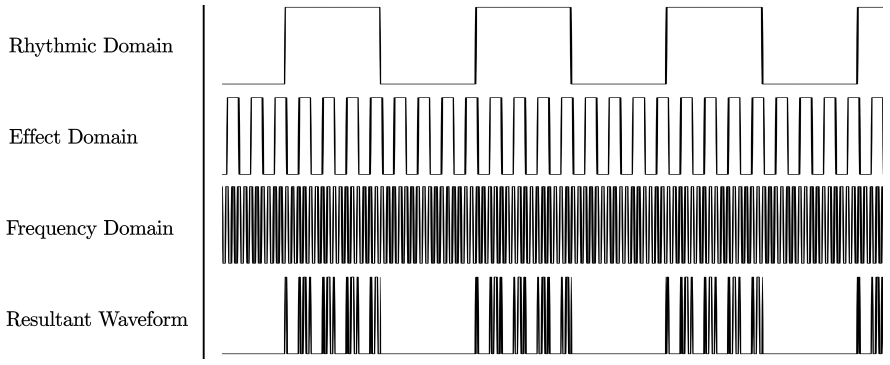


Figure 17: An illustration of temporal orders as individual pulse waves, modulating lower order waves. Functionally, these waveforms can be seen as mixed via a logical AND (&) operator. The duty cycle is 50% to clearly demonstrate the seemingly fractal properties of this arrangement. Not to scale. An audio example of this image can be heard at `https://doi.org/10.5258/SOTON/D1387`.

The previous `square.c` program outlines the basic foundation to generating 1-bit music, however the implicit application of this routine belies its potential to describe the foundation of structural musical operations (rhythm, metre, timbre) in a 1-bit sonic environment. One can consider further structural additions to perform these same operations but at progressively slower time domains. The `fractal.c` program (Listing 7) is an expansion of this exemplar and a practical implementation of Figure 17.

The program implements the simple, timer-driven oscillator model nested at each of Åkesson's time domains. The reader will notice the multiple appendages to the basic timer function defined in `square.c` are identical, however each now toggles a unique, enumerated 'virtual', or 'software', output rather than the I/O port directly. One might consider, at this point, the numerous `output` variables to be *conceptually* polyphonic; every output keeps track of a unique logic level. This is collapsed in the following, where each output is collectively applied as a bitwise mask to the previous: `PORTB = output1 & output2 & output3 & output4;`. The AND operator here extracts a subset of the bits in the state.

[68] Decrementing is always more efficient than incrementing on the AVR platform. [147]

```
#include <avr/io.h>

#define FREQUENCY  150
#define EFFECT     8
#define RHYTHMIC   1
#define STRUCTURAL 4

int main(void)
{
  DDRB = 0b00000001;

  uint8_t   output1            = 0,
            output2            = 0,
            output3            = 0,
            output4            = 0,
            pitch_counter      = 0,
            effect_counter     = 0,
            rhythmic_counter   = 0,
            structural_counter = 0;

  while(1)
  {

    PORTB = output1 & output2 & output3 & output4;

    if(pitch_counter-- == 0)
    {
      pitch_counter = FREQUENCY;
      output1 ^= 1;

      if(effect_counter-- == 0)
      {
        effect_counter = EFFECT;
        output2 ^= 1;

        if(rhythmic_counter-- == 0)
        {
          rhythmic_counter = RHYTHMIC;
          output3 ^= 1;

          if(structural_counter-- == 0)
          {
            structural_counter = STRUCTURAL;
            output4 ^= 1;
          }
        }
      }
    }
  }
}
```

Listing 7: Recursive square wave generator (`fractal.c`: `https://doi.org/10.5258/SOTON/D1387`).

As every oscillator is embedded iteratively within a series of conditional branches, the frequency of each oscillator gets progressively slower the deeper within the chain it resides. This is further regulated by a predefined variable, in this case named after the temporal domain it has been determined to operate at. The program does not necessarily have to be structured in this way, however it is advantageous for optimising speed of code execution. Rather than having to test multiple conditions frequently, operations that occur on a much slower scale can be invoked only when a single condition is true. The result could be considered a 'micro-composition': a growling tone repeats three times before pausing at equal durations[69].

This example is important as it succinctly demonstrates the topography of most routines, their temporal arrangements and also forms the basis of all musical code I have written thus far. It is noteworthy that these operations are not an artefact of technical requirement but a correspondent to psychoacoustics; they are, from the microcontroller's perspective, entirely arbitrary. Transposing this program through time domains would not change the relationships of resultant signals, nor the program's operation, but would dramatically affect the result to the human listener. Therefore, one might consider this paradigm to be 'clocked' two-dimensionally; dependent on both the speed of hardware and the persistence of human perception.

[69] See `https://doi.org/10.5258/SOTON/D1387`

3.2 Micro Music Macro Language

3.2.1 Why Music Macro Language?

Micro Music Macro Language (MMML or, preferably, μMML, when the text-mode allows it) is a musical description language, and reinterpretation of *Music Macro Language* (MML) and drives the primary 1-bit sound routine (`mmml.c`) for this project[70]. The core, *raison d'être*, of μMML however, is that it is (as the name suggests) primarily intended for use with microcontrollers[71]. This particular manifestation is a distillation of numerous approaches to low memory composition and a methodical 'coagulation' of strategies, fragments from previous routines and experimentation.

MML is a music scripting language designed by SHARP Corporation in 1978 [149] often embedded as a command in corporation, or platform specific, *BASIC* idiolects[150]. The entry for MML in the book *Beyond MIDI*, a compilation of languages and codes to represent music digitally, dates MML to the early 1980s with its first iteration included in Yamaha consumer computers [150]. This conflict may be symptomatic of the *ad-hoc* nature of MML; instances of the earliest MML instructions in SHARP machines were referred to as simply MUSIC [151] or MEDLY [152] commands, obfuscating the chronology.

MML is, principally, a method of sequencing higher-abstraction musical events, such as pitch, rhythm and form, rather than concerning itself with synthesis and I/O operations. MML is patently and centrally musical; that is it inherits its symbology from traditional musical models, chiefly music notation. Interestingly, MML is remarkably similar to another descriptive notation by Chris Walshaw called *ABC* or *ABC Notation* [153]. ABC is inspired by the written traditions of Irish folk music and devised entirely independently of MML [154][72]. Although syntatically comparable, the semantics of the naming convention are indicative of intention and, consequently, function. Simply, Music Macro Language is a *language*, ABC Notation is a *notation*; ABC is designed to describe; MML is designed to instruct.

Historically, MML was more commonly used by Japanese composers than their Western contemporaries [155, 156, 150], further evidenced by the Japanese origins of popular MML software tools, such as the *3ML editor* [157] and the *PPMCK MML compiler* for the Nintendo Entertainment System [158]. Presumably, the endemicity is due to the prevalence of SHARP [159] and NEC home computer systems in Japanese markets [150] (the NEC line of products being one of the best selling and numerous computers in 1980s Japan [151]). The language has, however, seen recent popularity in the Western chipmusic scene [160, 161, 162] likely due to the availability of accessible MML compilers for the NES and other legacy video game hardware [163]; hardware which reached a global market. Much of the material concerning MML is unfortunately in Japanese, inaccessible to a monoglot such as myself, and I, like many other English speakers [158, 155] have relied on the few translated documents

[70] If one wishes, they can see a prototype of this approach in the companion `spooky.c` program (`https://doi.org/10.5258/SOTON/D1387`).

[71] And, by extension, *microcomputers*. This term has, however, been disfavoured in computing nomenclature for *personal computer* (PC) [148]. Therefore using 'micro' to refer to a PC, implies that the speaker is referring to machines of the 80s and early 90s. This inference is, coincidentally, well suited to μMML's intended purpose; see *Cross-Platform Implementations*.

[72] Perhaps an example of 'evolutionary convergence' in software development?

available in ageing Internet repositories[73]. That said, the language has no authoritative texts, nor any form of standardisation. Implementations of MML rely on common practice alone; applications potentially presenting variations in syntax, commands and features. Despite the confusion, this has positive consequences: the language can evolve to suit the needs of the composer or, more commonly, the platform. Micro Music Macro Language is yet another of these MML pidgin dialects.

During the development of μMML I have borrowed and synthesised ideas from a few key sources: the aforementioned PPMCK [158], the MML entry by Matsushima in Beyond MIDI [150] and the execution in the MML workstation, 3ML [157][74]. The power of MML, and thus μMML, comes from its *interpreted* nature. Unlike formats such as MIDI, which requires[75] explicit declaration of any command, valid MML can exist as a single note name assuming durations, timbres and octaves from previous declarations or default parameters. MML is in no way dependent on any systems that generate audio (and could be considered a software scheduler) however, pragmatically, the language is defined by the instruments it is intended to control. In the case of Micro Music Macro Language, this is a set of 1-bit, sonic operations.

Ultimately, I have chosen to closely follow the MML paradigm as I have identified it to be a more efficient solution to storing musical data than the typical pattern/chain structure often found in software sequencers and routines [165, 166, 167]. Formats such as MOD expect four bytes per note [166], whereas μMML is less rigid, requiring only one byte per instruction for the most frequently used commands. This means that identical data does not have to be declared repeatedly, such as 'volume'[76] or current octave and, where these values are changed every note, is no larger than verbose declarations. I had initially taken the popular pattern/chain approach in a variety of early test routines, all of which were unsuccessful, either more limited or much larger in size than μMML. The `spooky.c` program, although less than one kilobyte in size, is more inefficient and less flexible than `mmml.c`[77]. Pattern lengths are fixed, requiring declaration of data per *museme*[78] which, in this case, would mean as many bytes as there are semiquavers. The greatest strength of MML in this environment is its flexibility. Whereas the *spooky.c* (and similar) routines [79] limit the user to a particular style of composition, this approach is far more adaptable as there are comparatively fewer limits to note duration, pulse width and, especially, metre. Routines like this that rely on fixed array lengths allow ease of computation (as the program knows when to end a passage and how to store them), but limit all music made within that program to a fixed event length. If the user wanted 14 instead of 16 musical events per bar, two bytes would go wasted per pattern. Over multiple channels and numerous patterns, one can quick extrapolate and calculate how much memory this approach might consume. The data organization

[73] Most of the seminal English material on MML has been referenced in this paragraph; it is exceedingly sparse when compared to MIDI, for example.

[74] This influence technically extends to *Mabinogi* [164], the 2004, massively multiplayer online role-playing game. The 3ML MML integrated development environment (IDE) has been designed to allow users to create custom music for the game's simple implementation of MML. I have not played Mabinogi, so I am not sure how the language is expressed in-game.

[75] This could be construed as a false comparison; MIDI is not a language, but a data format. Even so, if one were to write MIDI by hand, one would find that it is not fundamentally different in nature.

[76] In our case, volume is pulse width. See 2.2.

[77] See `https://doi.org/10.5258/SOTON/D1387` for `spooky.c` program.

[78] The smallest declarable unit of any musical framework. See 4.1 for a detailed definition.

[79] See: `https://doi.org/10.5258/SOTON/D1387`.

methodology implemented here could be likened to the *hypertracker*[67] framework, allowing for less literal playback of musical events and more structural nuance. Material can be looped at any point in the sequence as metre is dependant on the user's compositional language entirely and can differ between channels. Essentially, parameters are waiting to be told what they should do, rather than expecting to be told at every event. Consequently variables, such as volume, can be set once and left many note events before being defined again.

3.2.2 The μMML AVR Implementation

The μMML adaptation is born out of a personal love for the musicality, simplicity and practicality of MML and its surprising efficiency when applied to low memory composition; especially when comparing MML (and, by extension, μMML) to other file formats. Additionally, due to the human readability of the source code (the syntax is similar to traditional notation) and abstraction from the hardware, MML seems a very convenient way to get others involved in the project.

Consider the following section as a debriefing on how data is processed and expressed by the `mmml.c` program[80], with examination of the rationale behind some of the more arbitrary design choices. Developing a 1-bit routine is an imaginative endeavour; although there are restrictions on what is both possible and practical, many, very musical, choices are left solely to the programmer's discretion: the number of software channels, the method of polyphony, the level of control over pulse widths, the temporal resolution (or museme, see 4.1) to name just a few. These decisions affect the total possible compositions in any given routine: if one designs a routine to be particularly individualistic, its products will sound of the same ilk, if one designs a routine to accommodate a diverse, broad range of approaches, the code may bloat with increased use-cases and sacrifice efficiency. This 'tug-of-war' between elements is what makes the 1-bit routine as expressive as the music it produces — the discussion as to whether different DAWs affect the composer's process is trivial in comparison! Largely, it is in the arbitrary that we can discern the most creative meaning.

This section is important in documenting how one might approach programming a routine, by analysing each decision that has been made in the creation of `mmml.c`. Should the reader wish to build their own I remind them that, as long as the basics outlined in Section 3.1 form the foundation of their approach, anything mentioned in Section 2.2.2 and 2.2.3 is eminently possible (and more). It cannot be emphasised enough that the composition process begins here. Whilst anyone may compose using existing 1-bit routines, to some extent, any resultant artefacts can be considered collaboration between programmer and composer. In summary, the choices I have made at this stage of the project dramatically affect the resulting artistic outcomes.

The `mmml.c` program is a four channel routine: three channels of melodic, 1-bit pulse waves and a simple, percussive PWM sampler. Each channel is labelled A–D respectively and all are mixed via the pulse interleaving method. The composer has the choice before compiling to replace the sampler (channel D) with a percussive noise generator, which (currently) cannot be dynamically toggled in software.

There are eight pulse waves available: 50%, 25%, 12.5%, 6.25%, 3.125%, 1.5625%, 0.78125% and 0.390625%. These have been selected due to both simplicity of implementation (the waveform peak is defined by enumerating the frequency, divided by powers of two: `waveform = frequency >>` n`;`) and because they are the most dissimilar timbrally. As mentioned in Section 2.2.2, the spectral image can be described by $m = \frac{1}{pw}$, where pw is the given pulse width (expressed as a percentage) and every nmth harmonic ($n = 1 \ldots k$) will be missed.

[80] Microcontroller code is available here: `https://doi.org/10.5258/SOTON/D1387`, DOS port available here: `https://doi.org/10.5258/SOTON/D1387` and GCC wave builder available here: `https://doi.org/10.5258/SOTON/D1387`.

It presents the composer with the ability to perceptually vary volume as well as timbre. The small number of duty cycles is also related to the datatype instructing the duty cycle command (explained later on).

With this routine I have attempted to maximise stylistic flexibility as to allow multiple, disparate compositional methodologies to be executed per microcontroller. This said, the four channel approach is biased to a set of particular instrumental techniques, reminiscent of the NES [107] (which I find most compelling). Whilst more channels are certainly possible, this will be at an increasing cost: further chunks of data must be allocated for each new channel, so must additional oscillator code which will expand the base memory footprint and slow the routine. This would be less costly if the routine employed PPM, however I opted for PIM so that the maximum range of timbres could be utilised. The inclusion of a PWM sample channel (or percussive noise generator) is certainly a large down-payment of both memory (samples must be stored in program memory) and computational time[81], but drum writing is typically dependent on repetition[82] (or at least, reliance on repetition is not as noticeable or detrimental to the composition as it might be melodically [168]) so that, when composition begins, the likelihood is that it will not expand as rapidly as melodic channels.

The basic operation is outlined in Figure 18. There are two distinct sections: synthesis and sequencing. The synthesis portion of code calculates four output samples before checking if the 'tick' counter has reached zero. In this context, a tick (terminology borrowed from MIDI) is considered to be the smallest unit of *musical* time, or the museme (see 4.1). Perched at the extremes of the Rhythmic Domain, a tick can last anywhere between 16320 to 64 samples depending on the chosen tempo[83]. In μMML, this is considered a 128th note which is the smallest, directly addressable duration. At very low tempi (equivalent to a very high BPM, tempi in μMML is faster at lower values of `t`) a tick could easily migrate away from the Rhythmic Domain completely, even into the Pitch Domain[84].

The decision to set the minimal duration of accessible time to 128th notes influences which type of compositional methodologies can, straightforwardly, be practised via the system. The routine was not chiefly designed to modulate notes on the sub-Effect, Frequency or Sample Domain scales. Although it is certainly possible, and this type of use-case was considered when building the routine — it requires an out-of-paradigm approach[85]. When this is considered, in tandem with the fact that notes must complete their specified duration before the respective channel can be manipulated, the goal of the routine becomes clear, and thus the intentions of the composer[86]. The routine is designed to compose from the 'top-down', that is, abstract away the waveform generation (delegating as much of Frequency and Effect Domain event processing as feasible to software) and orchestrate musical events from the

[81] PWM sampler code is not as efficient to implement as tone generation code — at least, I have not found a solution that can be expressed as efficiently.

[82] Discussed further in Section 4.3.

[83] Calculated by logically bit-shifting left (little-endian) an 8-bit value by 4, as per line 327: `tick_speed = buffer3 << 4;` (the four being a fixed value chosen to make best use of the 8-bit range), multiplied by the number of samples generated in the synthesis loop (one per channel).

[84] Lower tempi translate to higher BPMs due to the increased proximity of smaller values to zero, thus resetting at a higher rate when decremented.

[85] For example, one might decrease the tempo variable so that a tick occurs at the Frequency Domain. This transforms MML notes commands so that durations might be doubled, trebled, or otherwise increased in value, granting access to 256th notes, 512th notes and so on. This is addressed under the 'Colour Within The Lines' heading, found in Section 4.3.

[86] Me.

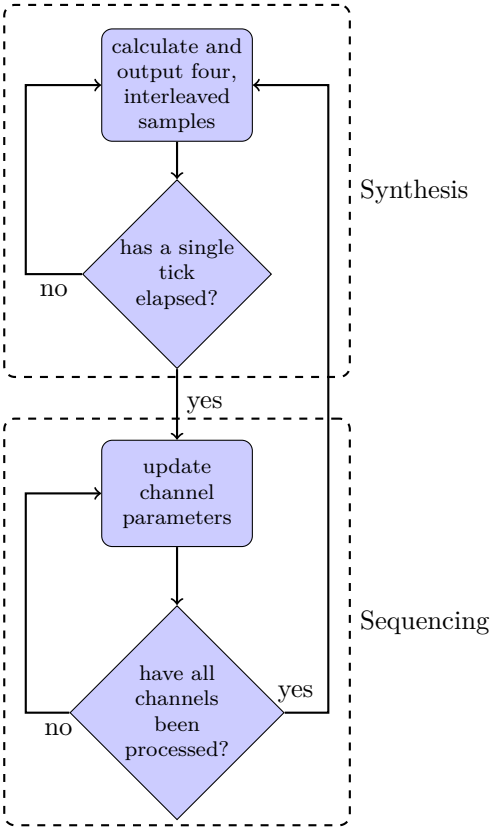


Figure 18: A flow diagram of the `mmml.c` program's basic structure. The labelled areas demarcate at which temporal resolution the enclosed operations occur: the synthesis is at the frequency domain and the sequencing area is at the very edge of the Rhythmic Domain. The routine is clocked by whatever method is best for the platform. To make the most of the microcontroller's abilities, `mmml.c` requires the entire 8MHz and is running as fast as the microcontroller - no clocking required!

Rhythmic Domain. There is no way to sequence sub-Rhythmic, timbral evolution without 'cutting' notes into command-value pairings and inserting a timbre command in-between (e.g. dividing a crotchet into semiquavers and inserting commands between each respectively).

The compiled bytecode (read from the `musicdata.h` file — generated by the μMML compiler[87]) is largely structured as follows:

```
        ______BYTE_____
       |               |
BITS : [0000]     [0000]
FUNC : [COMMAND]  [VALUE]
```

Each byte is split into two 'nibbles': four bit values with a possible range of 0–16. Each nibble requires a second, defining the value of the command. This is not *entirely* abstracted from in the human readable language, but it is key to understanding why some variables are limited to a maximum of 16 states. The language is structured so that the most frequent commands required are represented by the smallest data type interpreted by `mmml.c`. In a few cases, this structure is appended with an additional byte, as below:

[87] See: `https://doi.org/10.5258/SOTON/D1387` for GCC/microcontroller version and `https://doi.org/10.5258/SOTON/D1387` for DOS version.

```
        ______BYTE_____           __BYTE__
       |               |         |        |
BITS : [0000]     [0000]         [00000000]
FUNC : [COMMAND]  [IDENTIFIER]   [VALUE]
```

This behaviour is required by the *Function* command (see Table 2), where there is always a trailing byte (technically a two-byte value). Where higher precision is required, the nibble datatype, capable of representing numbers from 0 - 15, is not sufficient. Thus the Function command uses the structure of a general command as a generic 'flag', capable of specifying sixteen additional functions and indicating whether the next byte in data should be interpreted as a new command, or an extension of the previous.

There are four main chunks of data for each channel (a sequence of commands dictating how each individual channel should behave, see Table 2), plus an additional block of data for each macro (channel agnostic sequences, referable in a channel's 'main chunk' of data). These are stored contiguously in a single, one-dimensional `unsigned char` array called `data`. The structure of this array is stored in a companion, `unsigned int` array called `data_index` which, as the name suggests, holds an index of where each chunk of data begins. The first four chunks are always channels A, B, C and D respectively, then macros 1, 2, 3... etc. It is unhelpful to imagine a specific maximum length of events for individual channels, it is dependent on the global size of all channels and macros combined, where any data beyond 65535 cannot be indexed (due to limitations of the declared array datatype).

µMML	BYTECODE	COMMAND		µMML	BYTECODE	COMMAND	
r	0000	rest		g	1000	note - g	
c	0001	note - c		g+	1001	note - g#	
c+	0010	note - c#		a	1010	note - a	
d	0011	note - d		a+	1011	note - a#	
d+	0100	note - d#		b	1100	note - b	
e	0101	note - e		o,<,>	1101	octave	
f	0110	note - f		v	1110	volume	
f+	0111	note - f#		[,],m,t,@	1111	function	<----

µMML	BYTECODE	READS NEXT BYTE?	COMMAND	<----
[	0000	yes	loop start	
]	0001	yes	loop end	
m	0010	yes	macro	
t	0011	yes	tempo	
	....	n/a	0100 - 1110 unused	
@	1111	no	channel/macro end	

Table 2: A table listing all possible commands in the core music data read by the `mmml.c` routine, alongside their evocation values. The 'value' field lists the first nibble in each byte; the program then expects a further trailing number between 0 — 15. In the case of the *function* command (1111, or 0xF), this trailing value must be one of those listed in the lower table. `mmml.c` then requires an additional byte, allowing values from 0 — 255.

The language interpreted by the μMML compiler deviates slightly from the nibble structure

required by the `mmml.c` program and syntactically follows that of traditional MML very closely. As μMML is essentially a more legible version of the music data interpreted by `mmml.c`, it is possibly more useful to describe the behaviour of the commands outlined in Table 2 via the human readable analogue. The following is a crash course in μMML which may not be entirely comprehensive, but is required to understand the notation system that will be used hereon.

Pitch/Rest & Duration (0000 — 1100)

The most elementary building blocks of μMML is the note-duration paring. Notes are declared by specifying the pitch name in lowercase[88] (`r` for a rest) with a trailing duration. Duration is notated as the denominator of the note subdivision in common time. For example, the note C lasting a crotchet in duration is written as: `c4`. Durations can be 'dotted', in a fashion identical to classical notation, continuing the duration by half of the note's original length. This is simply appended to the end of the note-duration paring as such: `c4.`[89]. Stating a duration is not required for every note; once a duration has been declared, the compiler will assume that every subsequent note without a duration declaration is the same in length as the most recent duration declared. Thus, the passage: `c4 e g b g16 e` is equivalent to: `c4 e4 g4 b4 g16 e16` (the bytecode will be the same either way).

To sharpen a note, either `#` or `+` may be appended to the pitch. For example, one would notate an F♯major scale as per the following: `f+ g+ a+ b c+ d+ e+ f+`. Notice that the enharmonic `e+` is a valid note.[90]

Octave (1101)

Unlike notation systems such as MIDI (where note values exist on a range between 0 and 127 [169]), μMML does not encode information about octave in the pitch declaration; it is specified separately. As changes in octave are not as frequent as changes in pitch within a single octave, memory can be saved by using a smaller datatype (in this case, nibbles, as there are only twelve possible pitches in μMML[91]) to represent pitches, and a further nibble to represent octave. In both cases, each declaration is actually a byte in length: the first nibble dictates function (is note, is octave, etc.) and the second the value (between 0 — 15)[92].

[88] Currently the compiler is case sensitive, I might change this in future.

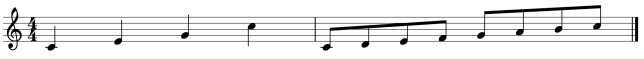
[89] Currently, this is functionally identical to `c4 c8`; as there are no note-on attack transients (unless created manually with `v` commands) the waveform will continue unbroken across both note events.

[90] As of writing there is no compiler support for flattening a note, therefore one must substitute an enharmonic equivalent. For example, D♭major must be instead notated as C♯major: `c+ d+ f f+ g+ a+ b + c+`.

[91] This *technically* correct, but suggests immutable tuning to the twelve tone scale. Whilst I have chosen to tune `mmml.c` to equal temperament, there is nothing preventing the reader from changing the tuning of each of these notes to whatever they wish. The reader should be aware, however, that tuning systems with more than twelve notes will require omissions as these cannot be addressed. Perhaps in a future version of μMML I might include microtonal notes - it would certainly be interesting!

[92] To get the same operable range as MIDI, the current 4-bit datatype would require an additional three bits — which sounds like a trivial addition (a whole byte is seemingly saved) but, due to the frequency of notes to octave changes, the memory increase is significant. Using `4000ad.mmml` as an example, there are 990 octave commands and 4452 note commands, sizing equally as many bytes. If three additional bits were appended to each note command, 990 bytes of octave commands would be replaced by an additional 37.5% of all note commands, or 1669 bytes total. This trade-off increases program size by approximately 68.58%. To get the same efficiency (again, using note frequencies in `4000ad.mmml`), octaves must consume less than an additional 22.24% of all note commands, which could only be achieved by adding a single extra bit, but would only represent just over two octaves of notes. This is yet a further example of how intended compositional

Octaves behave in a fashion similar to note durations, in that they only need to be defined once; the compiler (and routine) will assume that each subsequent note is of the same octave. Octaves can be defined in three ways: by stating the octave literally, as in `o3`, where the third octave is picked[93] or relative, using `<` for a relative octave decrease, and `>` for a relative octave increase. For instance, in the above example, the `>` command 'jumps' up one octave to `o4` from `o3`, which was the last octave defined. The compiler converts relative octave commands to literal octave declarations, so no memory is lost or saved by using any of the commands. The relative jumps are there to aid composition in μMML and make code more legible. Octaves currently may only range from 1 — 5; higher octaves have not been included as they will incur tuning errors (as mentioned below). Thus, the μMML analogue for:



is notated as:

```
o3 c4 e g >c <c8 d e f g a b >c
```

When writing low-memory, low-speed music, there is a key problem when synthesising (western) pitched waveforms: consistent, equal tunings. Because a counter (more specifically: 'countdown timer') must either be decremented (or incremented) until it reaches a precise value, the resolution of this counter is important. To generate higher notes, faster amplitudinal changes are required, consequently software parameters must be updated more frequently. The higher the frequency of software events, the smaller the number the timer must count to. The Western twelve-tone system has some particularly specific relationships between notes that require a high precision timer to accurately generate. As calculations with large numbers is computationally intensive on slower systems with 8-bit architectures, high fidelity timers will generate lower notes. Larger temporal resolutions require more possible states per second. To get these notes into an operable range, smaller numbers and data types must be employed; the Attiny simply cannot decrement a number from 60000 to 0 fast enough at 4MHz to produce an audible frequency. Unfortunately, as a consequence of this, notes will be progressively further 'out of tune' as the frequency increases - that is, they will deviate from their equality on a logarithmic scale. Figure 19 illustrates how, with smaller integers, tunings become progressively divergent from their absolute values due to an increasing loss of precision.

Of course, the above is certainly not platform agnostic and, depending on architecture, different problems will be encountered and alternate concessions must be made, but these issues are still relevant to the investigation as the spirit of the practice holds true whatever the implementation. Ultimately this notion of implementational limitation is important as technical compromises will influence subsequent compositions. As a consequence of Figure 19 for example, higher notes have been avoided (less obvious in 8MHz routines, as calculations

practice will dictate the best way to construct a routine. It also has some silly compositional consequences, where picking a key that is less likely to require octave changes will actually *save* space (e.g. one might wish to consider how frequently a melody or figure moves over the octave divide to reduce the number of bytes needed to represent it).

[93] μMML is entirely dependent on the system running `mmml.c` and is not really clocked to any specific tunings. This means octaves and notes are always relative to whatever the operational frequency of the routine is. At 8MHz, the frequency of the project's music boxes, o3 is around middle C.

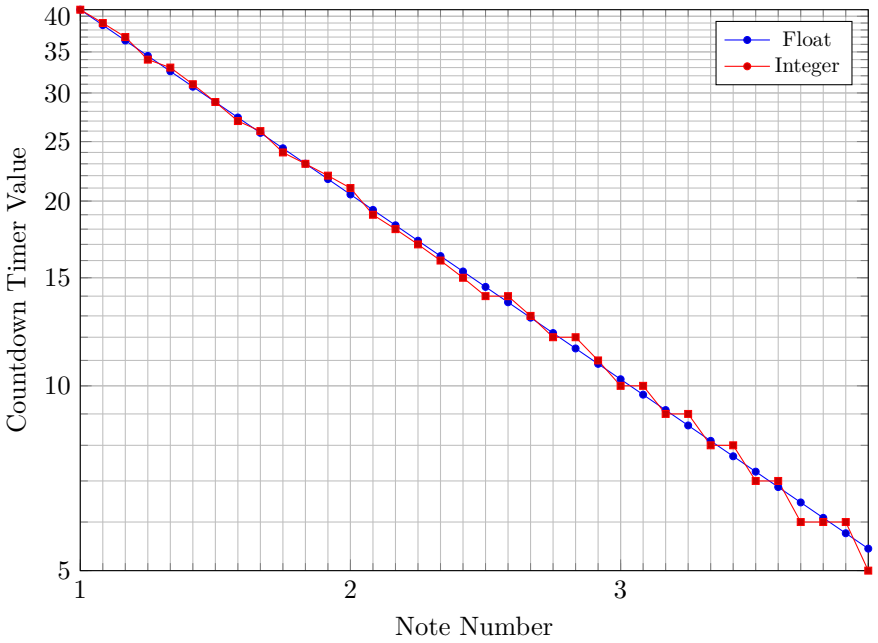


Figure 19: A comparison of integer and floating point data type precisions on twelve-tone, equal temperament tunings as the frequency increases. The X axis labels indicate successive octaves starting from an arbitrary pitch. Note that the countdown timer does not indicate any particular frequency; resultant frequency is contingent on the rate of decrementation. Pitches were generated using the `notefreq.c` program, which is powered by the equation: $f = \frac{b}{d\sqrt[12]{2}}$, where, b is the base note counter value (in this case, 41), d is the distance to the base counter value in semitones, f is the frequency of the note d semitones away from b[106, 170].

have more temporal headroom, and more pronounced in those at 4MHz, such as the `spooky.c` routine) and, where high notes have been used, these are not sustained to minimise perceptibility.

Volume (1110)

Volume is the last of the core functions — those that require only a byte to be declared including the accompanying value. Obviously in a 1-bit environment, volume refers to pulse width, thus timbre and volume are conflated to the `v` command. As mentioned previously, there are eight possible duty cycles which can be called by specifying a value between zero and eight, where zero is silence and eight is a 50% duty cycle. Judicious use of this command allows for a fantastic range of possible timbres and instrumental expression. For example, Figure 2.2.2 (shown in Section 2.2.2) might be created using the sequence `o3 v5 c32 v4c v3 c8 v2c v1c r4` for the first sound, and `o3 v1 c32 v2c v3c v4c v5c v4c8 v3c v2c v1c` for the second. Notice the order in which the volume commands have been placed to create the respective envelopes, or volume 'ramps'.[94]

[94]This example highlights that the current version of μMML is far from perfect, there are a few key modifications that would save a significant amount data in some use-cases. The most significant of this is the planned 'instrument macro'. Essentially this is a secondary macro function (operating mostly at the Effect Domain), which instructs the program to apply a sequence of volume alterations over note durations. This changes transients without having to split the note into multiple separate instances, interspersed with volume commands. For example, the prolonged note in line 127 of `goose-communications.mmml` notated as follows: `v1 e32 v2e v3e v4e v5e v6e` might instead have a macro definition at the beginning of the document, such as this: `@i v1r32 v2r v3r v4r v5r v6r`, then an in-line notation like this: `i1 c8.` (where `i` represents 'instrument'). Although longer to define initially, if there were multiple inline declarations of this envelope, data savings would be cumulative. Essentially, one might imagine this as defining a set

The actual numerical value of the volume command is reversed in software and a '1' will represent the loudest volume. This is because the number attached to the volume command dictates how much the current frequency should be bitshifted to the right — or divided by two. Dividing the frequency by two once will result in a square wave, dividing by two twice results in a duty cycle of 25%, dividing by two three times, 12.5%, and so on. I realised that, like human hearing, the perceptual change in volume due to a reduction in pulse-width is approximately logarithmic. Intermediate volume commands seemed to add no real sonic variation but those in powers of two did. Rather than design some complex, memory intensive function to scale 0 — 15 to all interesting pulse widths, I settled for eight different volumes/timbres and a small, easily implemented function.

Contiguous Material Loop (11110000 & 11110001)

The composer may loop contiguous material, specifically those commands that are in a continuous, unbroken sequence, by inserting a loop start point, loop end point and number of times the inner material should repeat. Looped material should be enclosed in square brackets with the loop number immediately following the open bracket. For example, the arpeggio in: `[8 c8 e g >c <]` will repeat eight times. The loop number may be a number between 2 and 255. The comparatively larger range to those commands previously listed is due to the command type; as described, the 'function' command (1111) allows for a trailing 8-bit value[95].

Loops may be nested; for example `[2 [8 c4 ]]` will repeat `c4` sixteen times. There is an upper limit on how many repetitions are possible; this value is defined in the header of the `mmml.c` program.

Macro (11110010), Channels & Channel End (11111111)

Music is typically repetitive in nature; it relies on the human tendency to recognise patterns. Consequently, one may often wish to repeat sections of material multiple times within a piece. The loop command *does* allow for this, but only when the material to be repeated is *contiguous*, which is rare. Consequently, any material that returns in a piece verbatim, but separated by unique information, cannot be reused. The macro command solves this problem

of 'instruments', akin to saving patches on a synthesiser. The instrument macro could also include octave switches or relative pitch movements, allowing for automatic transposition (discussed below). As to why this was not included in the current build, firstly an additional 'slur', or join, command would be required, instructing the program to prolong the volume macro across two (or multiple) note commands. As there is a fixed set of durations that can be literally declared, non-standard note durations (for example a double dotted crotched) will be interpreted by `mmml.c` as two separate notes. This is not an issue in the current iteration as notes cannot be affected over their duration, thus would be superfluous. (I somewhat anticipated that I might include this feature, especially in later compositions, so I have left the `&` symbol where a note is held over a non-standard duration). This also helps with legibility during subsequent analysis. This is not interpreted by the compiler.) Furthermore, potential redundancy reduced the attractiveness of the feature. Those more involved pieces (such as `4000ad.mmml`, `goose-communications.mmml` or `paganinis-been-at-the-bins.mmml`) change their instrumentation frequently, with very few repetitions of verbatim expressions. This may lead to the proposed instrument macro being underused, or perhpas used too frequently (with too may unique macros declared), thus consuming *more* space or (more likely) not saving enough memory to justify the initial downpayment.

The value of a feature such as this is dependent on how much material will make use of it. μMML was designed specifically with consideration to the program memory size of the Attiny45/85. With only four or eight kilobytes (respectively) to store both routine and music, if a piece is unlikely to use a potential feature then it should not be included; this data may be entirely wasted and might be better used for more musical material.

[95] A value of 1 would perform the material once, thus negate the need for a loop. A value of 0 would not play the material at all, for which, again, a loop would not be required.

by inserting a symbol where the material should be played. This symbol represents a string of data outside of the four channel structure that the data pointer jumps to, plays, then returns to read the next command in the original channel's data. To select the desired macro, two bytes are required: the first, instructs the program to treat the second as a number. This number is a pointer to an index in the `data_index[]` table and the channel data pointer will move to the defined location in the `data[]` array.

The layout of an μMML project should be structured as follows:

```
@ - oscillator A data
@ - oscillator B data
@ - oscillator C data
@ - sampler data
@ - macro #1
@ - macro #2
@ - macro #3
... and so on.
```

Each `@` symbol declares the end of the previous chunk of data with a channel end command (11111111). The first four `@` symbols are the 'home' chunks of data and the last are individual, voice agnostic nuggets of musical material that can be sequenced within the 'home' voice material. The channel command indicates which channel the following material should be placed in. Unlike most MML implementations, material can't be split across the document. Once the end of a channel is specified, the next begins. The last channel declared will always be a macro channel. The order in which material is placed in the first three channels is somewhat irrelevant as the oscillator generation code is homogeneous. Channels are stored contiguously in an array and are demarcated by the channel end command. This is inserted automatically after each channel's material has ended. As this is left up to the compiler, it does not have to be declared in the composer's μMML code. The compiler counts the bytes and generates an index where the `mmml.c` program can find where each of the respective materials begin.

In the current version, macros cannot be nested. Only one index is stored when jumping out of the main routine. A new macro command encountered while venturing out of the original channel's data stream will overwrite the original saved position with a new one. This means that, when this new material is finished, the data pointer will jump back to the previous macro playing then, once it reaches the channel end command, it will interpret this as the end of the original channel and loop back to the start of that channel's material. This could be avoided by using a two dimensional array for storing pointers: the X axis would be channel and the Y would store the locations departed in successive macros. This might be useful but, at the moment, I have had no real use for it. It seems that music is generally repetitive, but only to a certain extent; too much variation arises to warrant the extra few hundred or so bytes to implement such a system[96].

One must be cognisant when using octaves within macros and remember that the octave jump shorthand is a literal declaration of the current octave. Improper declaration of octaves may result in passages where the octave jumps erratically. The compiler will treat this in

[96] Although, it should be noted that this exact method to nest loops.

relation to the last octave used — probably somewhere at the end of the third channel (as the fourth is just a sampler). To avoid this, the desired octave should be stated at the start of each macro's material[97].

Tempo (11110011)

The tempo command is defined using `t` and sets the note playback speed only. These speeds do not correlate to any specific BPM and are defined arbitrarily; both by the internal clock and however long it takes to execute the code. As of writing I have not calculated BPMs, and these will vary based on architecture and clock. When read by `mmml.c`, this value adjusts the `tick_speed` variable.

Tips For Reducing Filesize In μMML

The compiler is not intelligent and does not have a 'compress composition' function. This certainly would be a useful feature, especially if it intelligently employed those techniques outlined in Section 4. Until then, this must be performed manually. The information here is related to that explored in *'Colour Within The Lines'*, found in Section 4.3, where the peculiarities of any system ultimately influence and subsequent compositions. Here are some methods for dealing this particular system:

- Repeats should be used to save space only when the material to be repeated is larger than four bytes. As the repeat function requires three bytes to implement, it is uneconomical when the material to be looped is shorter when declared literally. For example, the following: `c8 c8 c8` (compiled to `0b00010011,0b00010011,0b00010011`) would be a byte larger when written like this: `[3 c8]` (compiled to `0b11110000,0b00000011,0b00010011,0b11111111`).
- Similarly to the above, calling a macro has a minimum footprint of five bytes: two for the declaration within the channel data, two for the sixteen bit number that indexes its location and one for its channel end command. Additionally, it is almost guaranteed that an octave command will be required at the start of the macro and when returning to the main loop, adding a further two bytes. Therefore a macro will generally fill four bytes for a declaration and three bytes for each invocation.
- Care should be taken to avoid redundant notation, for example repeated durations. For example `r16 r16` can be replaced with `r8`, or `c4 c8` should be notated by a `c4.` command.

These suggestions will save a few bytes at a time, but this is cumulative! The total savings over a whole piece could allow for additional melodic/rhythmic variation, or instrumental expression — so pedantry is eminently forgiveable!

Efficiency is imperative in this endeavour; uneconomical scripting on machines with lower clock speeds directly translates to a more limited functional praxis. For all code listed herein a significant amount of fine-tuning has been undertaken to try and find the most efficacious solution. There have been many iterations of even single line functions to maximise the microcontroller's output, increasing memory and computational economy.

[97] Yes, you can share material between the sampler and the pulse channels. It's utterly pointless but you can do it.

These optimisations range from managing the idiosyncrasies of Atmel's AVR machine code (decrementing variables rather than incrementing) to imaginatively negotiating hardware limitations, often involving careful counting of clock cycles. This becomes somewhat challenging when working in C; the AVR GCC compiler can seem to take on a life of its own when compiling to machine code! When working close to the metal, saving even a single clock cycle can potentially allow for more oscillator channels, higher attainable notes or more 'palatable' tunings.

4 Compositional Approaches

4.1 Introduction

The Internet is littered with fragmentary tips and suggestions on how to best optimise composition for certain platforms. For example, the guide to the *Cross Platform Music Compiler Kit* (XPMCK) includes tips on optimising code depending on compiled platform [171]. This said, I am yet to encounter a comprehensive guide to the compositional systems and, ultimately, mindset required to successfully reduce file size and retain aesthetic interest. The following section documents the techniques and strategies I have encountered, employed, researched and developed in producing the companion portfolio of compositions for Attiny13/45/85.

Practical experimentation, in the form of exploratory composition, has been an integral part of the investigation process to address the primary research question of how using 1-bit synthesis within a low memory environment can inform or change compositional methodology. Whilst it may be possible to derive an answer to these problems algorithmically[98], without composition, there is not only creative value but efficacy in heuristics. It is immensely difficult to identify the most efficient solution if no solution is ever undertaken practically. Synthesising the tools, frameworks and concepts addressed in the first part of this investigation, the following chapter documents the application of these concepts through composition.

Importance is placed on achieving a low bytes per second (B/s): a simple calculation which divides the total filesize by the duration of the resultant piece. This is certainly not a comprehensive figure, it does not include piece complexity, nor musical interest, but it does provide a criterion by which different approaches can be easily judged and contrasted for relative success[99]. The nature of low memory composition is to keep the bytes per second per piece as small as feasibly possible, but retain musical interest.

Compositional decisions undertaken when creating the music in this portfolio are inevitably informed by my compositional inclinations, musical tastes and previous artistic experiences. Musical interest, for me, is related to a piece's complexity over time: if a piece is too simple for too long, it rapidly becomes boring, if a piece is too complex for too long, it becomes incoherent. The exact method by which this is evaluated is beyond the scope of this project so, for now, I will use my compositional inclination and subjective taste. Cutchfield proposes [172, 173] a relationship between complexity and 'randomness', plotted in Figure 20 as complexity against Shannon's H (essentially randomness).

Music at the peak of this curve is complicated, but self-referential. Music tends to have an internal 'dictionary' embedded within the structure of a compositional work. This

[98] This is rather the musicologist's 'philosopher's stone'; a piece of software that might describe a piece's harmony, melody and construction so that an automatic analysis can be generated. It seems so simple in conception, but in practice it is somewhat difficult!

[99] The routine size has been explicitly ignored from all B/s quoted in this section, referring to the compiled size of the μMML code alone. This is because, once the initial routine down-payment has been made, any further expansion is based on the size of subsequent μMML code. I shall consider the B/s with routine included in Section 5

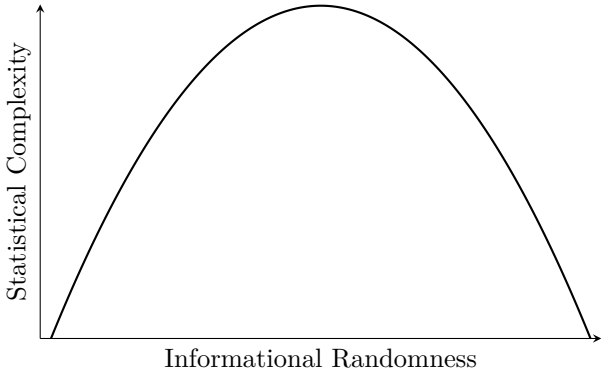


Figure 20: Crutchfield's demonstration of the relationship between complexity against entropy. This is a crude reduction, but is sufficient to adequately demonstrate the core concept. As the informational system gets more entropic, the complexity increases before waning and returning to zero. On the far left of the graph would sit a composition of a few notes repeated infinitely, and on the far right might be the product of a simple 'randomize' note function.

reference is built from repetition of musical ideas; as Ball explains "we make sense of what we hear by framing it in the context of what we have heard already" [174]. The audience is made familiar with structures presented to them (built from components such as harmony, genre and context) and, to a pleasurable effect, these structures are altered once established, defying expectation and expanding the working grammar of the internal logic of the composition. Music that defies repetition, either completely, or relies heavily on the listener's pre-existing understanding of theory alone (however casual this may be), may sound confusing and inherently less 'musical' [175][100] The top of Crutchfield's curve is practically rather hard to compress, yet it seems to be where the most interesting music lies. For example, music located on the far left of the complexity curve is relatively straightforward to compress; a simple piece consisting exclusively of a repeated short phrase, or tone, can be executed in just a few bytes. Equally, music located on the far right of the curve, that which is most musically complex, might be generated with a short randomise function[101].

Curiously the relationship between complexity and musical enjoyment shares the same 'inverted U' function [176]. Therefore, if we are to write interesting music that belies the medium, we must find intelligent compositional strategies that communicate an intermediate musical complexity (even an illusion of such is adequate) for as long as possible within minimal program memory.

Ultimately, for me, artistic expression should surmount technical 'pedantry'. There is always, however, a negotiation between memory costs and aesthetic benefit; a piece consisting of a single note is a complete technical success in terms of B/s, but (arguably) a failure when considering compositional sophistication and sustained listener interest over an extended duration. This section provides the reader with an explanation of how one might resolve these dilemmas and ensure that each piece is as small as possible, all whilst retaining novelty and eluding musical lethargy.

[100] It might be interesting to compose a piece that relies on as little pre-existing semantic inferences or semiology as possible, meaning instead contingent on tautological, *internal* definitions and rules.

[101] Some randomise functions can be quite complicated, but something akin to `noise = noise ^ noise >> 1; noise++;` (the noise generator used in `mmml.c`) would suffice.

4.2 Observations On The Nature Of Low-Memory Composition

The following concepts are important in explaining the creative motivations (and dogmas) of the low-memory composer (and help to contextualise those solutions proposed in Section 4.3). It is important that the reader understands the framing and informal rules by which one generally abides when constructing music in this fashion. These observations do not apply to 1-bit music in particular, but chipmusic composition in general. This said, 1-bit music is a sub-genre of chiptune for a reason; the ethos is closely aligned and the compositional techniques have considerable overlap. 1-bit music is a good solution for low-memory musical frameworks, but does not *have* to be low-memory in its execution. For example, Benjamin Oliver's 1-bit piece *Mr. Turquoise Synth* uses a 1-bit synthesiser controlled by a computer via MIDI [177]. When executed in this way, the composition can be as large, complex and non-repetitious as the composer desires, despite retaining a similar sonic aesthetic. Furthermore, with this approach, all techniques described in Section 2 can be employed without application of the strategies listed hereupon.

Forced Cohesion

Although there are methods of making pulse waves sound more, or less, palatable, the constraints placed on the low-memory composer engenders (oxymoronically) both timbral homogeneity and idiosyncrasy. Where the composer, unfettered by technological limitations, is afforded multiple instruments, synthesisers and sampled sounds, they may be entirely (and infinitely) liberal with their spectral palette. It requires either an enforced restraint (for example, only having clarinettists available for a recording session/performance), or arbitrary restriction (for example, choosing to write for clarinets exclusively) to curtail timbral breadth. The low-memory composer, on the other hand, must best (and perspicaciously) use whatever sounds are available, which results in recycling as often as possible (see *Recycling And Re-purposing* below). This is especially noticeable on systems that depend on sampled audio, such as the Super Nintendo Entertainment System (SNES) and Nintendo 64, where audio routines are generally reliant on short instrumental loops[102]. Memory restrictions on these systems limit the possible instrumental scope, forcing unusual ensembles and soundworlds. The SNES game *The Legend Of Zelda: A Link To The Past* [178] makes use of only thirteen samples,[103] which are used repeatedly throughout the game's soundtrack. Memory limitations demanded that the composer could not to wander and slip to new, disparate soundscapes over the duration of a soundtrack — enforcing homogeneity through memory limitation. This may appear creatively stifling, but may actually aid a piece — especially in the context of an album or soundtrack. By the use of a particular platform, the composer has automatic timbral cohesion. Humans are remarkably good at recognising and identifying pieces when presented with excerpts only a few tenths of a second in length [179]. This ability is not related to a piece's melody or hamony, but its timbre.

[102] Single wavetable samples that are looped, as opposed to loops of pre-recorded material a few bars in length (though this was possible and occasionally used).

[103] Twitter users @KungFuFurby and @jen_imago kindly examined the ROM for me (shout-outs to both of them; especially @jen_imago!) and found eighteen distinct samples, see the thread here: `https://twitter.com/jen_imago/status/1129052709930307584`. Within those eighteen sounds there are thirteen that are employed instrumentally: three brass-like instruments, two sounds approximating strings, a short sine-like waveform, a harp, a piano, a choir, a flute, a snare, a cymbal and a timpani.

Whilst this effect is certainly true of sampled audio can the same be said for 1-bit pulse waves? Whilst 1-bit music certainly benefits from forced cohesion, there is an upper limit to the decidedly unique articulations possible with 1-bit instrumental practice. That said, where sampled audio is used, or a repeated use of a particular instrumental technique, it does seem to grant the piece an individualistic timbral identity by simple repetition. The sound of Tim Follin's 1-bit ZX Spectrum routines are that of polyphony, extremely thin pulse waves, alterable 'volume' and noisy PIM mixing [114, 180, 181, 182]. Furthermore, as larger polyphonies grant Follin the ability to use block chords, alongside fluid decay transients, these techniques become ubiquitous, defining and consequently all compositions aesthetically cohere. For this, one might consider Follin's ZX Spectrum work as a 'micro-genre' — indeed this may extend to any set of works written in a particular routine. This is a huge creative advantage for those building their own music software, and certainly part of the personal allure of programming music routines for microcontroller. Pieces created with `mmml.c` have an aesthetic distinct from that of other 1-bit works. Although crude, the PWM sampler permits more prominent percussive material than other routines might. As there are few samples (due to memory restrictions) I do not doubt that the drum sounds used in μMML will be familiar to the listener by the end of this project — so much so that they would recognise them when presented alongside other 1-bit drum samples. The pulse widths addressable in `mmml.c` enable access to both timbral heterogeneity through larger pulse widths, as well as timbral homogeneity (and companion alterable volumes) bestowed by very thin widths. However, as only eight possible widths can be invoked, changes between them are quantised and can sound disjointed, unlike Follin's routines, for example. Furthermore, as `mmml.c` lacks the ability to continuously change pitch, glissandos and vibratos must be notated manually, thus, for this to sound convincing, these passages must always occur too rapidly for the listener to identify individual notes. Additionally, as there is no transpose function (due to the nature of how notes are stored), key changes are infrequent[104]. Although these 'features' may seem limiting, it is these deficiencies and imperfections, forcing idiosyncratic solutions that may not otherwise have been adopted through choice, that define sound of the medium.

Maximalism

One might consider chiptune (and this investigation) as minimalistic; this term referring not to the specific genre, or movement, in music, but to the ethos of simplification; the idea of "as simple as possible", "as little as possible" or "as small as possible" (the latter in software development). Seemingly in direct contrast to this 'minimalist' ethos, *maximalism* curiously emerges in the fundamentals of limited memory compositional practice. Maximalism is perfectly communicated by the "more is more" philosophy; an embracing of the excess [183]. My proposed expression of maximalism is perhaps a softer version of the composer Milton Babbit's, whose maximalist music includes (pretty wild) "all-combinatorial hexachords" and "all-interval rows" [184]. Even so, his definition non-intuitively applies here: "to make music as much as it can be, rather than as little as one can get away with". One might imagine maximalism as minimalism's antonym but, from my experience, when artistic foundations are cast in minimalist ideologies, the artist is free to push this framework to its limits. The examples are numerous: demoscene culture often favours aesthetically

[104] As they do not allow for easy recycling and Neckering, see *Recycling and Re-Purposing* and *Harmonic Neckering* in Section 4.3.

excessive, esoteric visuals to demonstrate coding capacity and the claustrophobic sound world of *black MIDI* compositions feed a single, wavetable sample[105] as much data as possible, making use of every available parameter[106] [185]. Whilst minimalism is indeed a large part of chipmusic — and low-memory composition as a whole — I would argue that minimalism describes only 'the container'. The endeavour is fundamentally maximalist: filling the metaphorical container with as much ostentatious, impactful and impressive content as possible to confute the common understanding of the container's limits. These limits may be a memory constraint, as with demoscene artefacts, or understood capabilities of platform, as demonstrated by Nancarrow's *Studies for Player Piano* [186]. So, while black MIDI may produce large file sizes (compared to a more 'typical' MIDI file), it has significantly more in common with this investigation than traditional definitions of musical minimalism. Additionally, most chipmusic is not *necessarily* celebrated for its small filesizes, instead successes in the face of constrained, seemingly rudimentary, aesthetic limitations.

Thus, the disposition of the maximalist sees unused memory as squandering potential: why forgo an extra few minutes of material, or the (ordinarily wasteful) addition of gratuitous variation, for memory that is not used? The character of perpetual *minimum* I have (potentially) obtruded upon this exploration — "as small as possible" — rewards brevity, encouraging the composer to complete their work at every point of potential expansion. The imposition instead of a *maximum* is far more constructive, dictating the hard boundaries in which to creatively expand. This is the usual scenario when writing in this manner; the limit is rarely fluid, more often strict and known in advance (sizecoding categories [2], removable storage media size, microcontroller program space [187], etc). This model allows the composer to more effectively *budget*; that is, critically evaluate which parts of their piece may consume more space than others.

Synthesised Versus Sampled

If spectacle was the primary element of consideration when producing low memory compositions, perhaps one might use all (or a significant portion of) available space to store sampled audio for playback. Indeed, this was occasionally the case for video games in the 80s and early 90s, where any sampled audio provided novelty (such as the soundtracks for *Skate Or Die II* [188] or *Blades Of Steel* [189] on the NES). However technically impressive a composition using this technique might be, a few seconds of compressed audio would not make a satisfying listening experience in the context of a larger work. It seems that humans want to be engaged by music for a significant period of time (i.e. longer than a few seconds) [190, 191, 192][107]. Therefore, this suggests a criteria of success related to the efficacy of the routine's bytes per second (B/s). Sampled audio will have a notably higher B/s than synthesised audio (if the initial memory down-payment of the sound routine is to be ignored), see Figure 21. Nevertheless, hypothetically, if the size of the sound routine is included, and

[105] Colloquially, black MIDI uses 'MIDI Piano' sounds. This is largely meaningless as MIDI defines no sampled audio, nor waveform generation, by itself. Often the term 'MIDI sounds' refer to the *Microsoft Wavetable Synthesizer* employed by Microsoft Windows to sonify MIDI files in the operating system's file explorer.

[106] Usually in the standard MIDI message paradigm: note, velocity, channel [169]

[107] Interestingly, these sources seem to suggest that the preferred popular song/album length may be related to the storage medium, seemingly related to maximalism? Or perhaps popular music is shaped by its commodification and commercialisation thus intimately related to its method of propagation? Maybe there is simply an upper limit on human attention spans for both the composer and the listener?

the sampled audio had a smaller footprint, the pre-recorded material would be aesthetically immutable[108]. When relying on such a technique, any additional sonic variation must be performed either generatively, via manipulation of existing sampled data, or simply the addition of data (uneconomical with respect to B/s). This suggests a crucial rationality for the synthesised solution: aesthetic plasticity. Flexibility and versatility in musical communication are the most important elements for sustained interest. The low-memory composer's dichotomy is the negotiation between compositional conservatism (reliance on simple, small sequences or looped passages) and introducing novel materials.

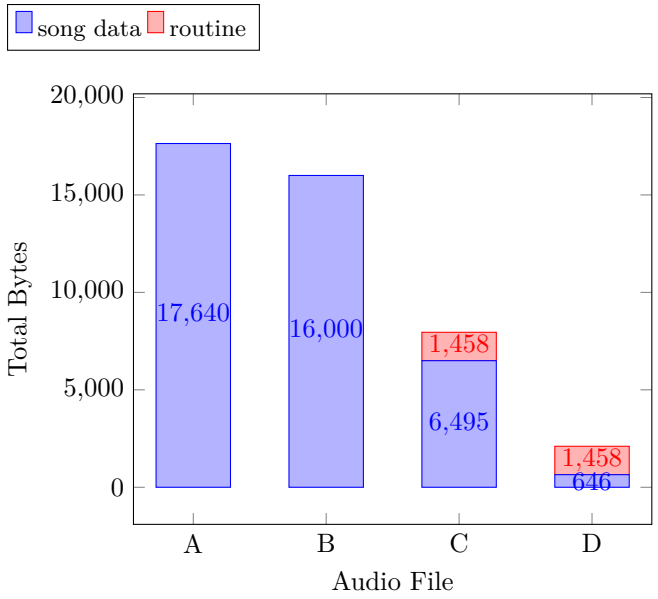


Figure 21: A graph comparing the size of various μMML pieces to popular sampled formats: A) 1 millisecond of 16bit Stereo 44.1kHz WAVE Audio, B) 1 second of MP3 audio at 128Kbps, C) `4000ad.mmml` (513 seconds), D) `jupiter.mmml` (730 seconds). For C) and D) the routines have not been directly included in the filesize calculation but appended as, when combining multiple `.mmml` documents, the routine size will only be included once. Additionally, *technically* sampled audio requires an interpreter for sonification, the equivalent of the `mmml.c` software. For example, one second of MP3 will require additional 30KB (approximately) for a minimal decoder [193]. With this in mind, if we compare `jupiter.mmml` to the CD-quality WAVE format, we can see that the μMML file is 99.9995% more effective.

Unless algorithmically generating audio (see 4.4), once song data has been compressed using the most efficient algorithms, it can be squashed no further. Where technology and mathematics fails to provide any further assistance, the composer may step in. The art of writing small music requires an intimate understanding of music theory and the judiciousness to know when to repeat (or remove unnecessary, *periphrastic*) materials.

Cheating The Medium

One can interpret "Cheating the medium" as exploiting the listener's expectations to hide a medium's technical limitations. The previous claims that interesting music must be complex, or inherently maximalist, could be met with disagreement and, instead, one may draw my attention to those minimalist pieces which rely on extremely gradual mutations or extreme repetition. This would certainly be a valid criticism and presents another, somewhat obvious, solution: hide behind genre limitations and compose unashamedly minimalist music for the

[108] One might create a routine that manipulated sampled data, as with many Amiga trackers [166], and even some ZX Spectrum 1-bit routines [128], but this would still be an example of a synthesised approach.

sake of minimalism! As the goal of writing constrained music is to belie the host medium's limitations, decisions that fundamentally alter compositional practice, but appear entirely arbitrary, distract the listener away from their potential purpose. The low-memory composer does not wish to labour the composition with flags to obvious concessions; a completely transparent process will confirm the listener's presuppositions and provide no expectational subversion. Whilst the platform (whatever this may be) is inseparable from the artefact, the mystery of the relationship between presupposed media capabilities, its limitations and a sophisticated artefact, has a somewhat performative element. Anecdotally, I find that in the demonstration of low-memory compositions, when explaining the compositional or technical tricks required to realise a piece, the piece is no longer judged on the merits of the composition, but instead by the process; any coherent music that emerges from this process is simply a bonus.

This suggests something intriguing and contradictory to the investigation's original premise. Although I am a strong believer that chipmusic can be separated from hardware — that the music can be enjoyed for the aesthetic qualities alone — it seems that, in the explicit endeavour of programming music where data is limited, one simply cannot isolate the product from platform. All techniques herein (both technical and compositional) are, pragmatically, memory saving techniques, but, in the context of this investigation, this may be an unintentional pretence. Perhaps the core difference between historical low-memory compositions and the current project is that working in these environments presents the auxiliary exercise of intentionally belying popular understanding of limitations, thus subverting expectations. The goal of much music is to present the listener with a compositional framework (either within the composition itself, or existing, cultural compositional tropes) then subvert those expectations to pleasurable effect [194]. There is a strong analogue in general sizecoding practice: perhaps one might imagine this process of belying limitations as a meta-music, or meta-composition, encapsulating the piece, providing an extra layer of interest.

The key difference between pieces such as `goose-communications.mmml` and `greatest-hat.mmml` is not just their compositional complexity, but the complexity of the `.mmml` file itself. The `goose-communications.mmml` file is packed full of volume changes, enveloping, super-fast arpeggios and attack transient octave jumps — extended chipmusic instrumental techniques, whereas `greatest-hat.mmml` focuses very much on counterpoint, three-part writing to adequately communicate intended harmony and, perhaps, more traditional, general compositional concerns. Although there is clearly a compositional benefit to adding instrumental ornamentation and expanding timbre, the primary reason for this practice is to make the piece more 'impressive'. The greater the instrumental repertoire, the more the piece appears to belie its medium.

Listing 8 is a transcription of Steve Reich's 'Piano Phase' in about 65 bytes of μMML (depending on whether the unused channels are included in the calculation). Other than adding some timbral effects, there is nothing more one could add to this composition to realise it any 'better'. It is entirely accurate in its translation from score to μMML and, thus, successful in attaining the original piece's compositional intentions. The minuscule resultant filesize demonstrates the immense efficacy of such a technique.

```
% channel a
@ t45 [99 m1 ] r2.

% channel b
@ m1 [96 m1 r128] m1 r2.

% channel c (unused)
@

% channel d (unused)
@

% macro #1
@ o3e32rf#rbr>c#rdr<f#rer>c#r<brf#r>drc#r
```

Listing 8: An μMML transcription of Steve Reich's *Piano Phase* (`piano-phase.mmml`), totalling only 65 bytes of program memory for around two minutes of music. Slightly shorter in duration than original piece, this adaptation could be extended by increasing the number of repetitions of a pattern before shifting phase (for example `@ t45 [99 [8 m1 ]]r2. @ m1 [96 [8 m1 r128 ]]m1 r2.`). As I am offsetting material by the smallest duration possible in μMML, I felt it disingenuous (and slightly boring) not to include a unique state per permutation. Listen here: `https://doi.org/10.5258/SOTON/D1387`.

4.3 Strategies For Reducing Compositional Footprint

This section documents the key techniques I have researched, developed and employed in my μMML work to reduce filesizes but retain compositional sophistication; ensuring that each piece sits towards the peak of Section 4.1's complexity curve. The aim is to make the low-memory compositional process transparent in order to demonstrate the construction (and rationale) of each of the works created as part of this research project.

It is useful to imagine the following approaches as organised *musematically*, or by *musemes* [195]. A museme, devised by Charles Seeger, is the smallest meaningful component of any musical syntactic framework, which Seeger defines as a single note or beat: the *tone-beat*. This definition works well in the traditional, *top-down* compositional approach (I use the terminology 'top-down' to refer to the pursuit of composition by traditional musical abstractions at the Rhythmic Domain — as opposed to *bottom-up*: concerning composition as a function of frequency over time) where the smallest unit of meaning is the pitch-duration pairing. The definition is seemingly simple but, from the musical architecture explored in Section 3.1, it suggests some challenging questions: what is the true museme, the musical-linguistic *atomus*? At what temporality does this framework cease to hold actual meaning (are there meaningful musemes at the sub-Sample Domain)? Defining the tone-beat as this smallest unit, implies a traditional, tonal compositional approach, where instrumentalists are instructed by rhythmic symbols, deputising more fundamental operations, such as timbre and waveform, to instrument and instrumentalist. Computational music allows for *intentional* organisation of meaningful musemes at ever smaller units of time. Conceivably, one might build a piece of timbral fragments, indeed Roads' Microsound [141] concerns the constructions of sound-worlds from sonic 'grains' — the purposeful arrangement of individual wavecycles at the Micro Domain. The etymological fog renders 'museme' as a fluid concept and, this flexibility, allows the museme's scale to change depending on the temporal scope

of the composition. This approach neatly frames the enquiry into efficacy and success (the smallest, most interesting music possible) by exploring strategies at different resolutions of museme.

I conceive of a process duality in computational music: raster and generative. In opposition to the forthcoming Section 4.4, in this section I am primarily concerned with *raster* composition: the placement of discrete, quantised data in a 'matrix structure'. Raster is used here as a metaphor; the terminology is borrowed from computer graphics, representing a (generally) rectangular grid of pixels. This methodology requires the intentional deceleration of quanta (in this case, placement of musemes on a grid) which is read by an interpreter. Products of this system will be a verbatim realisation of data declared in the matrix. We can consider this approach to be almost entirely *musematically* deterministic, as it has been pre-composed. 'Musematically' is important here as the indeterminacy is contingent on the resolution of museme. Higher abstracted musemes, such as the tone-beat, will surrender precise control of lower level functions to some auxiliary process (such as a human performer's instrumental technique, or a synthesizer's timbral programming).

Techniques for tone-beats rely heavily on traditional music theory (and wider musical practice) to make compositional 'shortcuts' — inferences of musical objects such as chords, scales and traditional melodic patterns. Although this is largely unavoidable when writing music, mutating textures or pitches in non-classical (or expected) ways may make creating more interesting low memory music more challenging. Composing with musemes at the sub-rhythmic level allows the composer access to sonic operations that can only be indirectly instructed in systems such as traditional score[109]. Concerning memory footprint, as the time-window has decreased, there are now more possible operations per second, requiring either *a lot* of data, or heavy reliance on generative functions and expansions of compressed sequences (not necessarily both). It is an intriguing world in which to compose, sitting somewhere between (human) composer autonomy and delegated control to a computational, compositional system.

This section aims to contextualise the compositional process of the μMML pieces through outlining specific techniques and strategies. I have, wherever possible, used these ideas within my music and, concomitantly, the pieces have informed the compositional strategies. There are four principal μMML pieces this project:

- `4000ad.mmml`[110]
- `paganinis-been-at-the-bins.mmml`
- `goose-communications.mmml`
- `jupiter.mmml`

The supplementary works provide examples of possible approaches, explore one particular facet of low-memory composition or serve to document progression. There are situations where I have not subscribed entirely to my own constraints-focused dogma — to the

[109] One may instruct a performer to use vibrato, but the exact sequence of micro pitch changes is left to the instrumentalist, or instrument. If one could sequence smaller steps of that vibrato's character, then the instructions would be entering the sub-Rhythmic Domain

[110] See: `https://doi.org/10.5258/SOTON/D1387`

detriment of filesize! There are many instances of memory gluttony, or gratuitous material expansion simply because I wished to investigate unfamiliar compositional ideas. `goose -communications.mmml` is an example of where I unrestrainedly, inefficiently allowed new material to enter the piece so that 1-bit techniques could be explored. Resultantly, compositions have frequently exceeded the aforementioned 1KB limit (ignoring the `mmml.c` routine size), but are instrumental in ascertaining those compositional techniques that dramatically reduce filesize and work musically, versus those which are either too inefficient, or simply too dull!

Consider the following strategies a practical manual for composing music suitable for low memory environments. Nearly all of these techniques are implementable by varying degrees of extremity: for example, one can observe *Technique 2: Recycling And Re-Purposing* and repeat musical materials infrequently, or base an entire piece around a single loop. Therefore, the sum total reliance on the techniques presented in this section is contingent on the use-case; if one has ample space in which to construct a piece, the below can be used to simply curb unnecessary expansion, if constraints are tight, one can adhere to the below uncompromisingly. It should be emphasised that these techniques, although they may have additional instrumentational influence, are focused on compositional construction. The explored techniques are:

1. **Slowing Down**: The result of reductive gamification of the low-memory endeavour — decreasing the tempo to increase B/s.
2. **Recycling and Re-Purposing**: The verbatim restatement of material and its effectiveness in saving space.
3. **Musical Decoys**: Distracting the listener from obvious concessions using prominent attentional lures such as a prominent melody or gratuitous solos.
4. **Melodies**: In addition to their function as decoy, how melodies can imply harmonic movement, taking some of the harmonic responsibility from other channels.
5. **Harmonic Neckering**: The use of musical superpositions: allowing a material to change its function based on that of companion materials.
6. **Colour Within The Lines**: Following best practice and making use of the functions available in a musical system. Moulding compositional ideas to fit snugly within the paradigm.
7. **Transposition**: Reusing materials at different pitches.
8. **Canons**: Imitative melodies that are offset by an arbitrary amount allowing for complex counterpoint with verbatim repetition.
9. **Phase Shifting**: Or tiny canons. Offsetting identical materials to create complex rhythmic interactions and new combinatory patterns.

The four major supporting pieces are a practical product of all of the techniques listed herein, executed to varying degrees. Although discussed within each section, the below roughly shows the various techniques each makes prominent use of:

	4000ad.mmml	paganinis-been-at-the-bins.mmml	goose-communications.mmml	jupiter.mmml
Slowing Down	✓			✓
Recycling and Re-Purposing	✓	✓	✓	✓
Musical Decoys	✓	✓	✓	
Melodies and Musical Presumptions		✓	✓	
Harmonic Neckering	✓			
Colour Within The Lines	✓	✓	✓	✓
Transposition				
Canons		✓		
Phase Shifting		✓		✓

Table 3: A table showing which compositional techniques are used in the four, primary supplementary compositional works.

Technique 1: Slowing Down

A somewhat spurious solution presents itself as a consequence of the conclusions drawn in *Synthesised Versus Sampled*: if keeping data transfer as low as possible is imperative, then simply reducing the tempo will nearly always achieve greater memory economy. Faster tempi are often less efficient as data will be processed at a greater rate[111] There are more generalised, less restrictive mechanisms by which memory can be saved, as numerous, slower pieces may fatigue the listener, but this strategy is still worth consideration. One possible response to this problem is to intersperse slower sections with faster to increase total runtime. Changing tempi in this way has the additional benefit of introducing musical variation; masking the composer's ulterior motive behind deliberate choice. The material labelled `% movement II intro (4/4)`, `% movement II main (4/4)` and `% movement II solo (4/4)` in `4000ad.mmml` extends the piece by around[112] 30 seconds, decreasing the bytes per second from 13.45B/s (without changes of tempo) to 12.66B/s. This increase in efficacy can, concomitantly, increase aesthetic diversity.

As a decrease in data size, or increase of duration, will positively affect bytes per second, the inverse will logically cause any piece to be less efficient. Considering the previous, where faster metres must be employed, reducing total duration, it can be surmised that more repetition (decreasing data size) would be required to get the same efficiency.

Technique 2: Recycling and Re-Purposing

Looping, composing a sequence so that the start and end of the material can be repeated without noticeable gaps, is possibly one of the oldest, and most prevalent method of saving memory in computational music [196]. Where systems such as the Commodore 64 had limited capacity to store data (with game cartridges as small as 10KB) the primary compositional method video game composers would utilise to mitigate limitations is repetition. Perhaps the reason this technique is so prevalent (and presents itself so immediately as a solution) is due to the fundamental repetitive nature of music [197]. Whilst the listener, in the context of video games, may be aware of the rational behind an endlessly looping piece, due to the use of the technique in wider music, an effectively written loop does not attract the listener's attention. If loops are not presented contiguously and infinitely, i.e. materials are presented again, reused internally within the wider composition, looping can be disguised as an aesthetic choice, distracting from the true intention for the repetition.

[111] Obviously, this is relative to the denominator of the metre. Simply, all durations of a piece could be halved (e.g. all crotchets translated to quavers) which would increase the tempo (achieving the same playback speed), but keep the data transfer identical.

[112] There is no strict tempo in μMML, because the routine is not clocked, the actual BPM depends on the hardware.

Examples of reused or re-purposed material in the investigation's compositional portfolio are too numerous, and fundamental to each piece's architecture, to comprehensively cite (as a demonstration, Figure 22 depicts the structure of `4000ad.mmml`, colouring all recycled material). It is simply a necessity in low memory music, so much so that two commands in μMML are dedicated to looping and recycling. The disposition is best communicated as thus: when composing, it is important for one to reflect on new materials and consider what a unique bar of material might add to the composition that could not be said with a straight copy of existing material. If the musical impact is not significant, then it does not need to consume memory. The piece `goblin-shark.mmml` is an example of poor low-memory composition. The piece's interest is reliant on unique variations on a theme, thus only 35.44% of the piece is recycled material. Consequently, `goblin-shark.mmml` has the worst B/s of all μMML pieces written for this investigation.

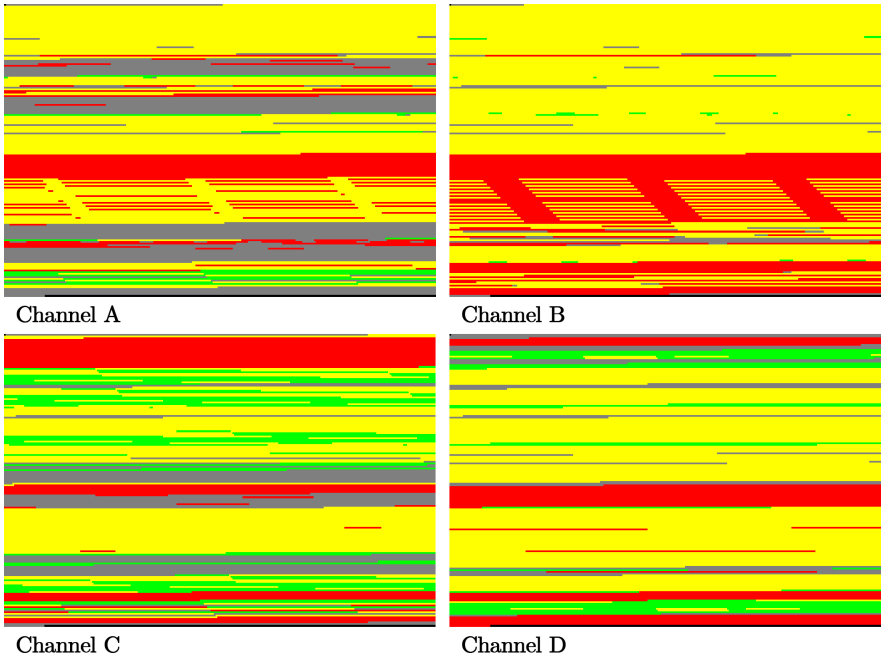


Figure 22: A visual representation of the repeated material in `4000ad.mmml`. The images are divided into four channels, depicting `mmml.c` channels A — D. Each pixel signifies every fourth 128th note, ordered top to bottom, left to right. Green pixels represent 128th notes that are looped, red pixels represent 128th notes that are part of a macro, yellow pixels represent 128th notes nested within both loops and macros, and grey pixels represent material that occurs only once (black pixels indicate that no data is present). In total (across all channels), the amount of reused material makes up 84.17% of the piece. Generated by the `1-bit-generator.c` program (`https://doi.org/10.5258/SOTON/D1387`).

One can see just how effective this technique is by comparing the total material recycled against the resultant piece's B/s, graphed in Figure 23. With an, albeit limited, set of data, there is an apparent negative correlation between how much of a piece is recycled versus its memory efficacy. This suggests that, the more a piece repeats its materials, the more efficiently it uses its data.

Technique 3: Musical Decoys

Overuse of recycled material will, eventually, foster musical inertia and, ultimately, tedium. Although repetition is a vital part of sustaining interest, after exposure to an extended period of identical cycles, the listener becomes desensitised, encouraging the brain to seek

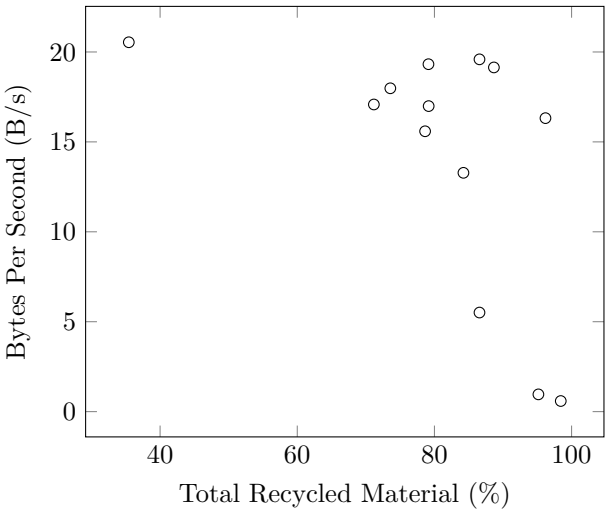


Figure 23: A graph plotting the total amount of material that is reused in a piece (calculated by checking, for each 'tick' (128th note in μMML), whether the channel pointer is within a loop or macro) against the resultant bytes per second (compiled size in bytes divided by duration in seconds). The B/s used does not include the routine, but including this will not affect the relationship between the data points as the compiled routine size is fixed (1458 bytes for the final `mmml.c`). Additionally, as multiple μMML pieces can be included in a program with only a single instance of the routine code, it is often unhelpful to add the extra kilobyte and a half to each piece's total size. The data is sourced from Table 5. The top-left point is `goblin-shark.mmml` (most inefficient) and the bottom-left, `jupiter.mmml` and `piano-phase.mmml` (most efficient).

new materials [198]. This human inclination works against the low-memory composer's principal tool of efficacy, instead promoting the introduction of novel materials and thus an increase in program size. One tactic of mitigation is to employ 'channel decoys'. Decoying in this context refers to attentional diversion — implementing prominent changes in one channel (or musical voice) to distract the listener's focus away from extreme repetition in another. More negotiation than complete negation, channel decoying affords the opportunity to extend the usefulness of materials through unadulterated, direct recycling.

In applications beyond saving memory, this technique is reminiscent of the interplay between vocalist and accompaniment in hip-hop, where the vocalist provides much of the musical variation against (often) a single, short instrumental loop. Whilst instrumental methods of distraction in hip-hop (variation of lyrics) are not literally possible in a low-memory environment, the fundamental concept is applicable. Moreover, the simplest method of executing this is by way of the notion of a 'beat'; a core rhythmic idea conveyed by percussive voices, effectuated in a manner similar to that of hip-hop. A repeated drum beat, or established groove, is extremely agreeable to the human listener [168], more so when it is played alongside additional variation in the wider arrangement. This behaviour is recurrent in constrained music, where single percussive figures can span the entire length, or majority, of a piece[113]. This can only be possible when the listener's attention is diverted by materials in other voices. For example, the reader may not have noticed that the drum solo at the beginning of `4000ad.mmml` (line 473, labelled `% drum solo Cmin7 (4/4)`) is re-purposed towards the end of the piece (line 552, labelled `% final stretch (4/4)`) as accompaniment. In the

[113]See soundtracks to *Super Mario Bros.* (NES) [199], *Legacy of the Wizard* (NES) [200] and *Pokemon Trading Card Game* (GB) [201] for an audio example, however there are numerous examples of this in early video game music.

second instance of the 'solo', the listener's attention is deliberately diverted from channel D to elaborate materials proffered in channels A, B and C.

A diversion works well if particularly novel, or frequently mutating; providing constant unfamiliarity. As such, an 'improvisatory' style solo, or virtuosic passage, can ultimately save space. Although this technique may require additional memory, any new material will be added to a single channel only, where it might otherwise expand data usage in multiple. Additionally, traditional compositional techniques such as a focus on melody may also provide an adequate distraction, encouraging the listener to engage with melodic changes, distracting from potential verbatim repetition in the accompaniment. This technique is employed repeatedly in `4000ad.mmml`, where materials for channels B, C and D are recycled (with some changes in channel B) in sections `% solo A (4/4)`, `% solo B (4/4)` and `% solo C (4/4)` over which channel A introduces a new 'improvisation' with each invocation. Equally, `til-there-was-you.mmml` uses the same technique, improving B/s by reusing materials in channels B, C and D as accompaniment for channel A's solo material (in the section labelled `% solo`).

Technique 4: Melodies and Musical Presumptions

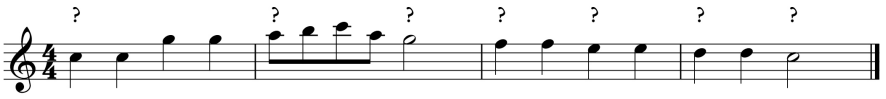
A solution prevalent, and preexisting, in wider musical practice, melody is a simple method of providing musical interest expanding only a single channel while material in others can be either looped or foregone altogether (see the intro to `paganinis-been-at-the-bins.mmml`). When paired with an accompaniment, melody can be considered a form of musical decoy, distracting the listener with unfamiliar material to mitigate musical malaise, accommodating repetition. Melody is a common solution to low-memory composition as even the simplest melody can simultaneously communicate harmony whilst providing musical form, thus prolonging interest[114]. One can imply verticality with the horizontal: for example, a horizontal sequence of `c8 e8 g8 >c8` communicates the same harmonic idea as $\begin{matrix}\text{C}\\\text{E}\\\text{G}\end{matrix}$, a C major chord. Many of the approaches taken in this investigation are built around melodies, as are most chipmusic works [115]. Monophonic voices seem to encourage compositional employment as 'leads', especially when the total number of voices are restricted. Indeed, one can fall back on the techniques of choral, and other small ensemble, composers; tried-and-tested approaches that remain popular compositional processes. Two, or three-part, writing is an apt solution for μMML, to the degree that works by Bach can easily be expressed by the language; a minimal transcription of *Præludium I* from J.S. Bach's *Das Wohltemperierte Klavier I* (`bach-prelude.mmml`) sizes only 661 bytes for 120 seconds of material (5.51B/s).

Monophonic musical excerpts from classical, or public domain, works were common in many early video games (for example, the soundtracks to Manic Miner [203] and Donkey Kong [204]); this is an effective strategy for the smallest musical implementations as invoking a well-known melody grants the piece harmonic context by listener association alone. Whilst this is a low-memory solution, one can imaging the storage as relocated from program

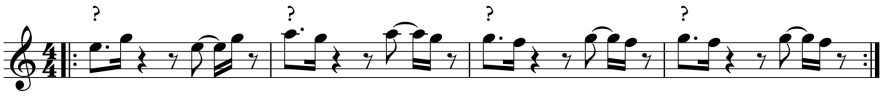
[114]The numerous monophonic works by J.S. Bach (*BWV 1001–1006*), Paganini (*24 Caprices for Solo Violin*) and many others are a testament to this technique's success.

[115]This claim is difficult to support concisely as there is no database of chipmusic pieces organised by whether or not they feature prominent melodies. In my experience however, it is usually the case that chipmusic pieces are reliant on melody — it is possibly the most immediate solution to writing music with monophonic voices. One can listen through the entirety of ZXART's music library [202] and will hear *many* compositions that are reliant on melody.

memory to the listener's musical experience. The following example demonstrates this effect practically (if the reader wishes to try this for themselves, see the footnote for answers[116]). If I were to ask a musician to complete the chords to the following (missing chords are indicated by question marks):



They will, most likely, correctly identify the piece and complete the passage. However, if one were to notate chords to the following, they would most likely struggle:[117]



There is simply too little harmonic information provided for the listener to ascertain an absolute harmony. To ameliorate this, the composer must supply context by either expanding the melody, or by adding an accompaniment, giving the listener more explicit harmonic cues. Obviously this comes with a collateral memory expansion.

Technique 5: Harmonic Neckering

'Neckering' here refers to the Necker cube optical illusion, where two simultaneous, valid interpretations of a cube's orientation can be inferred due to ambiguous depth cues [205]. In this case, these interpretations are harmonic in nature; materials are written to be as tonality agnostic as possible so that, with a small melodic cue, the supposition of key/chord is altered. This increases the possible use cases for the material, concomitantly increasing memory economy. For example, a guitar, in standard tuning, played entirely open produces a clear set of pitches, but an ambiguous harmony. The resultant chord E, A, D, G, B and E could be interpreted as an E minor 11 chord, or a G major six-nine chord (with the sixth in the bass), or perhaps a C major thirteenth, omitting the root (C). As if in harmonic *superposition*, all interpretations are equally valid (although some are, admittedly, more likely) until the chord is given harmonic context and the functionality becomes evident.

The simplest example of this can be seen in section `% theme A1 (3/4)` of `goose-communications.mmml` where the piece is deliberately composed to accommodate a pedal bass (label `% macro #06 - channel C: bass pedal C #1 (3/4)`). Pedals are easily repeatable and, due to their prevalence in wider music practice, do not draw attention to the low-memory composer's ulterior motive.

`4000ad.mmml` uses a more sophisticated Harmonic Neckering for the arpeggios in macros `m9` and `m10`, where the perceived harmony of the arpeggios is changed in their second evocation by the contrast against new material in channel C. Where `m9` and `m10`, played against material in channel C at line 319, create a C♯m^{11} chord, the introduction of `m17` (under label `% solo A (4/4)`) changes the apparent tonality of these macros to an F♯^{13}. Not only is this

[116] Chords for piece #1: C F C F C G^{7} C. Chords for piece #2: C^{add9} C^{13} F$^{6}_{9}$ Fm$^{6}_{9}$

[117] Unless familiar with the classic chipmusic album `BLUESHIFT`.

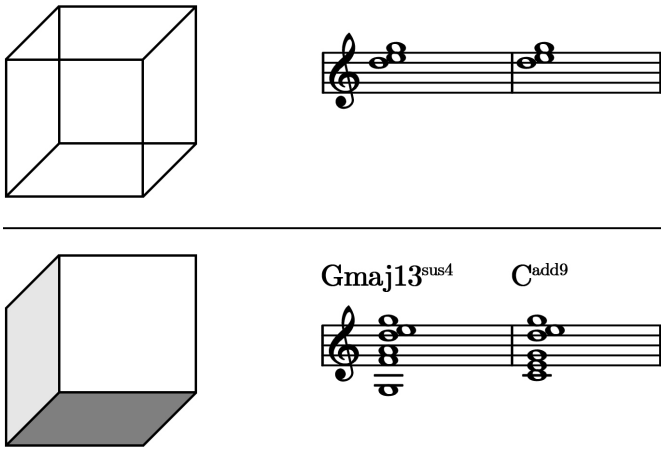


Figure 24: A visual demonstration of 'Harmonic Neckering'. Just as the orientation of the cube (top) is rendered unambiguous by further visual cues (bottom), the harmonic image can behave the same. Two, identical chords of D, E and G can become a pleasant cadence with further musical context.

aesthetically pleasing (harmonic expectations have been subverted) but it is also good low-memory practice. The introduction of a bass pattern in `m17` could simply restate the C♯m^{11} chord, however, because new data is being added anyway, the composer might as well use this opportunity to break from the established musical framework and introduce further variation (if relying on repetition to save space, occasions such as these can be rare!).

Mooning is terminology I apply to a specific flavour of Harmonic Neckering, conflated with channel decoying. Specifically, it refers to the use of especially long passages of *perpetuum mobile* ostinati and owes its (somewhat tongue-in-cheek) terminology to the infamous (at least, in the chipscene) Nintendo Entertainment piece *The Moon* (from the game *Ducktales* [206]), which featured the technique prominently. Specifically, repeated sequences are designed to busy a (usually) single voice with constant, easily repeatable harmonic content with the aim of producing alternate, extended, or completely different harmonies when contrasted against melodic content in other, simultaneous voices. Textural complexity can be largely delegated to the repeated figure, then simple movements of few musical instructions shift the apparent harmony and establish new tonalities. Consequently, as the alternative would be stating these chord changes verbatim in all channels, large amounts of memory can be saved under the guise of aesthetic choice. To be of greatest value, mooning will mostly employ Harmonic Neckering; ostinati that are too transparent or literal in their harmonic declaration will limit their possible application.

Mooning is used *heavily* in `goose-communications.mmml` where channel B is largely occupied by the ostinato in `m1`. The sequence: `@ o5 v6 e64 v5e32. v4<b64 v3b32. v6>a64 v5a32. v4e64 v3e32. v6b64 v5b32. v4a64 v3a32. v6e64 v5e32. v4b64 v3b32. v6<a64 v5a32. v4> e64 v3e32. v6<b64 v5b32. v4a64 v3a32.` is ostensibly a E^{sus4} arpeggio but takes the role of different harmonic extensions depending on the material it is contrasted against. For example, when played against material under label `% theme A1 (3/4)`, the chord sequence becomes E, D♭$^{6}_{9}$, A2, A2/G. However, when presented in section `% theme B1 (4/4)` this arpeggio becomes: E, EMaj^{9}/D♯, E^{7}/D, A^{add9}/A, Am$^{6}_{9}$/C, E/G♯, F♯^{7}/A♯, B^{7sus4}, B^{7} (largely due to the descending chromatic movement from E to A♯in the bass). Finally, it is presented

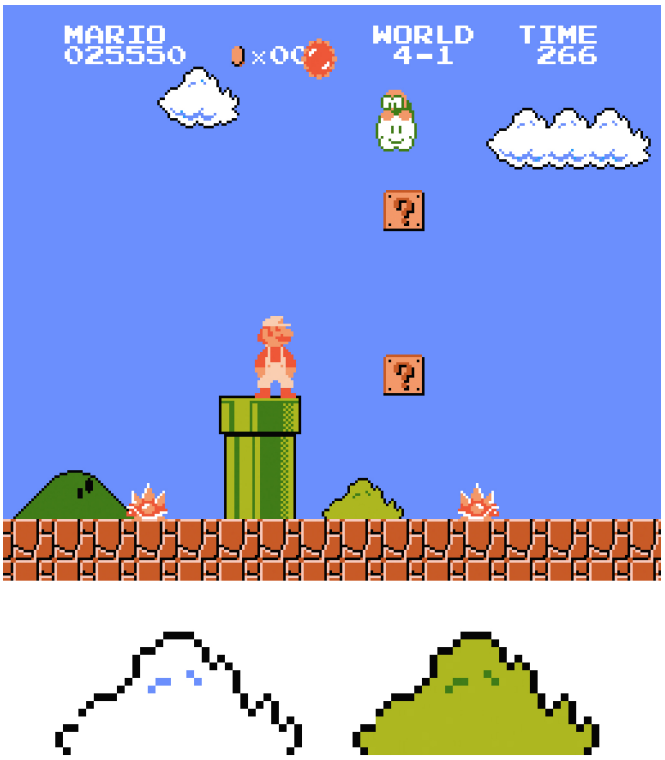


Figure 25: Examples of Neckering are not limited to music and are good practice for other low-memory artforms, such as graphics. In Super Mario Bros. (1985) [199], to save space, both the bushes and clouds share the same graphic, but different palettes are applied. Whilst the image is identical in shape, the perception of identity is altered by both position (proximity to the ground for bushes, height on screen for clouds) and colour - it has been visually Neckered in an analogous fashion to the musical materials described here. The screenshot is taken from: `https://en.wikipedia.org/wiki/Lakitu#/media/File:Lakitu_in_Super_Mario_Bros._(1985).png`

again over E, F♯^{9}/E, Am in section `% ending melody (3/4)`. The reader may notice that I have omitted to spell chords where these extensions would be dissonant; technically any statement of F♯^{7} underneath `m1` would result in F♯$^{11\sharp 9}$. Simply, I do not hear that dissonance as part of the tonality. It is interesting to me that this is even the case; if one removes the Neckering channel, the aforementioned chords do not appear with such viscerality. It seems that the brain is acting as a harmonic sieve, ignoring where the ostinato disagrees with the harmonic narrative and allowing the extension to contribute to the harmony when it is consonant. Whatever the underlying mechanisms, this is certainly exploitable by the low-memory composer!

Listing 9 demonstrates why Harmonic Neckering is such an powerful solution. Four methods of single channel accompaniment have been written to harmonise the following bassline: `o2 c2<bagfedg`. I have chosen a predicable chord sequence primarily for harmonic legibility; extended, unusual harmonies can be difficult to imply with two voices and relying on convention (the tropes of genre and wider music theory) can offset some of the responsibility placed on the composer to communicate musical objectives effectively. This is also true of the bassline; this endeavour may have been significantly more challenging if the bassline were a static pedal. The solutions somewhat delegate their own harmonic responsibilities to the movement of contrasting voices; without it, Neckering certainly would not work.

```
% channel a
@ t50 % set tempo

% harmonisation technique #1: arpeggio - 190 bytes
v7 o4 e64d v6 <g>c v5 [3 ed<g>c ] v4 [4 ed<g>c ]
v7 d<b v6 gd> v5 [3 d<bgd> ] v4 [4 d<bgd> ]
v7 c<g v6 ed> v5 [3 c<ged> ] v4 [12 c<ged> ]
v7 o3 af v6 dc v5 [3 afdc ] v4 [4 afdc ]
v7 bg v6 ed v5 [3 bged ] v4 [4 bged ]
v7 o4 c<a v6 f+d> v5 [3 c<af+d> ] v4 [4 c<af+d> ]
v7 o4 c<a v6 fd> v5 [3 c<afd> ] v4 [4 c<afd> ]

% harmonisation technique #2: harmonic neckering - 45 bytes
[8 o3 v5 c32 v3 d> v5 c< v3 c v5 g> v3 c< v5 c v3 g< v5 f> v3 c< v5 g
v3 f> v5 c< v3 g> v5 d v3 c ]

% harmonisation technique #3: melody - 38 bytes
v5 o4 e8degd4<b> | c8<b>ce<b4g | a8ga>c<g16rg8>ce16r | e8fdcd2

% harmonisation technique #4: simple harmonisation - 12 bytes
v4 o4 e2dc1<a2g>c<b

% channel b (bassline)
@ v4 [4 o2c2<bagfedg ]
```

Listing 9: Possible types of single voice harmonisation/accompaniment of a simple descending Ionian(ish) bassline in μMML, ordered by memory cost. Memory could be saved further in technique #3 by removing the 'echo' effect interleaved with the melody. See `https://doi.org/10.5258/SOTON/D1387` for an audio example.

Although the techniques are ordered by size of compiled code, they are also ordered by their timbral and harmonic interest. The correlation is not a coincidence; generally, the more commands added to alter timbre or nuance melodic content, the greater the aesthetic repayment, but the greater the filesize[118]. The first example (labelled `% harmonisation technique #1: arpeggio - 190 bytes`) is a near verbatim statement of the harmony by means of rapid arpeggiation. It is indubitably the most transparent method of communicating the composer's harmonic intention, but requires nearly sixteen times the memory than the simplest solution presented. In contrast, the simple harmonisation (`% harmonisation technique #3: melody - 38 bytes`) and melody (`% harmonisation technique #4: simple harmonisation - 12 bytes`) solutions are, although small, quite frankly, dull. Returning to the fundamentals of effective low-memory music, these solutions do not belie limitations (they sound both simple and uninvolved), they do not embrace maximalism and maximalist techniques (they present simple, sustained notes with no timbral variation), nor do they make best use of the 1-bit platform. For only seven more bytes than technique #3, the Neckering material has a more interesting compositional identity, creates extended harmonies in contrast with the bassline and pushes the timbre to the limits of the 1-bit platform.[119]

[118] At least in μMML. Some systems may utilise commands that invoke more complicated patterns. Generally though, unless deliberately mitigated by the routine, this holds true.

[119] A rather absurd thought has occurred to me: it *might* be possible to make a musical score Turing-complete through the use of loops and harmonic neckering. These harmonically ambiguous materials could to act as logic gates in a manner similar to that proposed in Changizi's paper on Turing-complete visual illusions [207]. For example, exploiting the fact that, when the listener is presented with a major or minor third, subsequent chords consisting of only the root and the fifth are interpreted as if they were the tonality of the initially stated note (A chord of just C and G will sound major if an E was played beforehand, or minor if an E♭was played). A NOT would be a parallel minor modulation, for instance. There are certainly potential problems with this approach, but it is an interesting thought!

Technique 6: 'Colour Within The Lines'

Colour within the lines, or: don't compose outside of the implementational paradigm. Using μMML as an example, a dotted crotched (`c4.`) requires a single byte to declare, whereas a double-dotted crotchet is not defined. It *is* possible to represent the double-dotted crotchet in μMML, however this necessitates an additional semiquaver command (`c4.c16`). Consequently, any instance of a double-dotted crotchet always requires two bytes to represent. The low-memory composer may wish to reconsider their compositional decisions to best fit the model; how important is the extra semiquaver? Might a dotted crotchet contraction, or minim extension, be a suitable exchange? If one were to transplant a MIDI file, with unquantised durations, verbatim into μMML, the memory footprint will almost certainly be significantly larger than that of a similar sounding composition which makes use of judicious note substitutions. These decisions are obviously not unique to note durations and may be more, or less, restrictive depending on the system. In short, one must endeavour to make best use of explicit commands and the features available.

This is not to say that one *cannot* operate outside the paradigm, in fact this may produce interesting results, but knowing when to 'bend the rules', and where one might compensate for any subsequent, potential memory increase, is important. Essentially, damage control for when non-standardised techniques are employed. As an example, to save space on the initial memory 'down-payment', potential functions were removed from the `mmml.c` routine, keeping the software as small as possible. The initial specification included an arpeggio, pitch sweep and vibrato function. Rather than exclude all undefined musical ornaments from future compositions, techniques can be incorporated using the existing commands. This will necessitate operations outside of the framework and will require extra memory. Most instances of vibrato I have used in μMML are essentially fast musical 'turns', where the semitone above, the note itself and the semitone below are played repeatedly and contiguously, notated as per the following: `c64 c+64 c64 <b64 >c64`. Examples of this approach can be seen in line 273 of `puppy-slug.mmml`, line 213 of `fly-me-to-the-moon.mmml` and line 39 of `4000ad.mmml`. In all listed occurrences, the vibrato has been extended so that it can be looped — a loop costs only three bytes to express. This is the aforementioned "compensation": if one wishes to tersely execute a vibrato, it should be ensured that the declaration is as small and repeatable as possible. One might wish to use: `c64 c+64 c64 <b64 >c64 c+64 c64 r32`, however this passage might be better written as: `[2 c64 c+64 c64 <b64 >]`. In this instance, only a single byte has been saved, even so, this is a good habit for the low-memory composer; the savings can be much more dramatic in other cases. This behaviour is another example of 'colouring within the lines', using the looping function to best effect.

Technique 7: Transposition

One technique that was not used in μMML[120] but prevalent in low-memory composition in general, is transposition [196, 165]. As notes are often synthesised in low-memory music (as opposed to sampled), they can be modified before, on, or after, playback. This flexibility allows musical fragments, or entire passages, to be transposed to new keys, thus

[120] Due to the method by which `mmml.c` interprets data, transposition requires interpreting the desired pitch shift in respect to both the current note and current octave. Additionally, there must be some mechanism by which transpositions can be sequenced (see the proposed instrument macro in Section 3.2).

prolonging a material's usefulness. One might imagine this to be another form of Neckering; materials that work best with this method are those which are as harmonically agnostic as possible. Basslines that hold a static note can be translated to consonance in any key and simple arpeggios can be transposed to create any chord of the same tonality — indeed, one can see how adding progressively more harmonic cues will gradually ossify a material's harmonic/melodic role.

This technique is employed in the `spooky.c` program. This routine is distinct from μMML as note data is represented by values 1 — 60, hence a simple additive (or subtractive) operation can be applied to note data, transposing patterns to different keys. The material labelled `// rhythm section main 0000` in `spooky.c` includes a simple bassline, moving in octaves, which is continuously reused throughout the piece and transposed underneath the melody, creating alternative harmonies in the process. As the notes are stored in binary (and thus difficult to read), to more clearly demonstrate, this technique is notated in Figure 26.

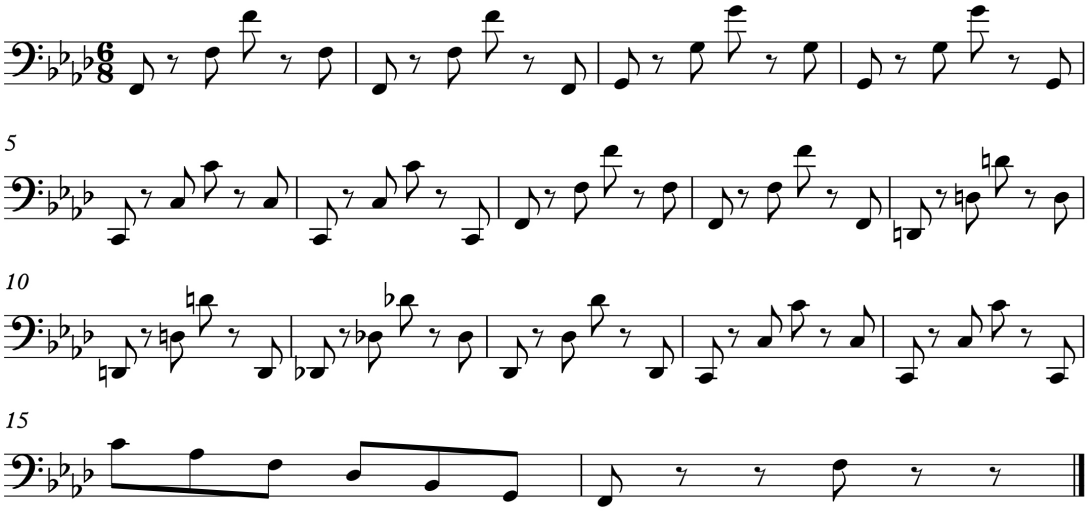


Figure 26: A scored demonstration of the transposed bassline in the `spooky.c` AVR program (`https://doi.org/10.5258/SOTON/D1387`).

Technique 8: Canons

With a comprehensive definition in the *Oxford Dictionary of Music*, canons are possibly the most effective of the straightforwardly traditional techniques discussed here. A canon is, in its strictest sense, musical imitation; material in one voice is repeated verbatim in another [208]. This repetition is often delayed by a bar, or any other duration, and can be translated to different octaves or starting on different intervals. When passages are reused literally, there only needs to be a single declaration for material that can populate the data for two voices; creating polyphony from monophony. To make this endeavour compositionally simpler, the use of short, repeated chord sequences ensures that material can be overlapped more frequently. Listing 10 is a short, two channel canon, 275 bytes in length. If we were to imagine that writing unique material for both channels would equate to, approximately, double the length of `m1`, then 246 bytes is saved by reusing material.

```
% channel a
@ [255 m1 ] % loop main melody

% channel b
@ r1 [255 m1 ] % loop main melody, but offset initially by a bar

% channel c (unused)
@

% channel d (unused)
@

% macro #1
@ % bar #1
  o3 v6g8v4g16v2g16v6>c16v2<g16v6>d16v2c16v6<f8v4f16v2f16v6>c16v2<f16v6>
      d32e32d16
  % bar #2
  v4<c16v2c16v4<e16v2e16>c16<e16v4>c16v2c16v4<g16v2g16v4f16v2f16v4a16v2a16v4
      >d16v2d16
  % bar #3
  v6>d32e16.c16v2e16v6<g16v2>c16v6<e16v2g16v6f16v2e16v6g16v2f16v6>c16v2<g16
      v6>d16v2c16
  % bar #4
  v4<c16v2c16v4<e16v2e16>c16<e16v4>c16v2c16v4<g+16v2g+16v4f16v2f16v4>e16v2
      e16v4d16v2d16
  % bar #5
  v6>e16v4e16v6c16v2e16v6e16v2c16v6g16v2g16v6f32g16.f16v2g16v6c16v2f16v6d16
      v2d16
  % bar #6
  v6<g16v4g16v6f16v2g16v6g16v2f16v6>c16v2<g16v6a+16v2>c16v6<a16v2a+16v6f16v2
      a16v6a16v2f16
  % bar #7
  v4c16v2c16v4<c16v2c16r16v1c16v4>c16v2c16v4<f16v2f16v4a16v2a16v4g16v2g16v4<
      g16v2g16
```

Listing 10: A short canon written in μMML. Only a single channel of information is provided (`m1`) and staggered by `r1` to create polyphony from monophony. See `https://doi.org/10.5258/SOTON/D1387` for an audio example.

If the technique is so effective, why have canons not been included in the four primary supplementary pieces? Simply, writing canons that are both interesting and effective is a somewhat difficult, slow process which becomes an increasingly challenging as more voices are added. This endeavour is further demanding in μMML as there are no semitone, nor octave, transposition functions[121], thus each canon must use identical materials.

However, strict canons can additionally make for an excellent delay, or 'reverb' effect. In lines 19, 88 and 144 of `paganinis-been-at-the-bins.mmml`, `m25` is played simultaneously, the latter two channels offset by a semiquaver and quaver, and reduced in volume to `v3` and `v2` respectively, creating a hall-like reverberation effect. Additionally lines 38 and 110 of the same file, delay identical material in two voices by a hemidemisemiquaver, producing a chorus/phaser. Although not in the spirit of the classical canon, it is practically identical and proves a good method of busying an unused channel with minimal data impact.

[121] The relative octave jump would seemingly work, but compiles to an absolute octave value, see Section 3.2.2. I will probably change this in future.

Rather than delaying the onset of material, what if two channels used the same materials, but different read directions? Essentially, the product of this is known as a *crab canon* [209, 210], where musical material in once voice is repeated in the second, only backwards. The savings for such a technique are essentially identical to the traditional canon, if not slightly larger, as a routine must be created to allow a second pointer to be reversed in transport direction. Figure 27 is a small composition demonstrating this technique across two channels.

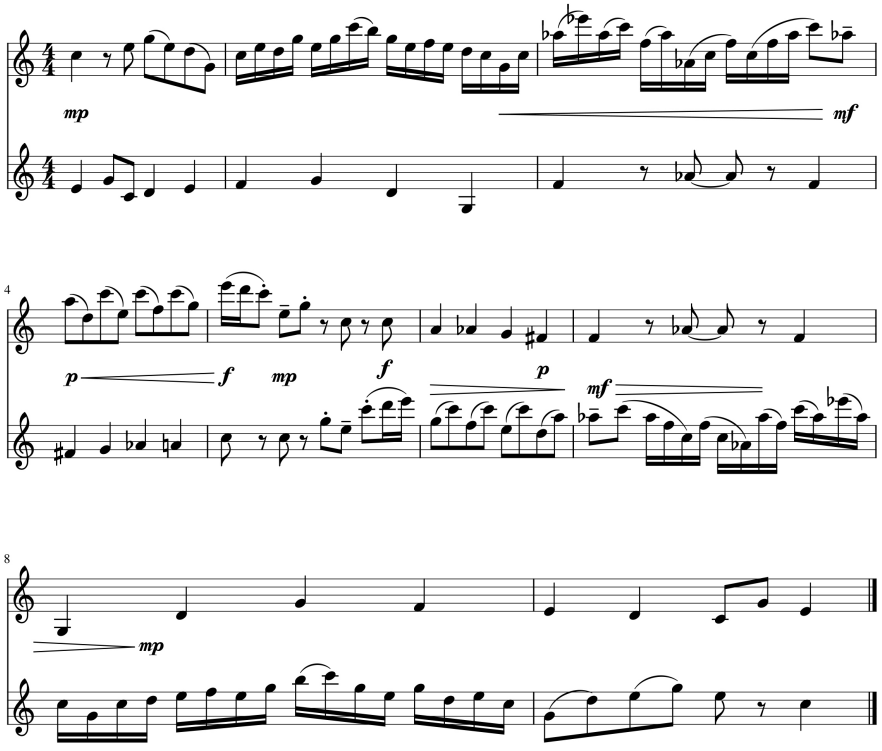


Figure 27: A short, scored piece demonstrating the *crab canon* technique. A crab canon is a compositional technique where two voices use the same material, only one voice presents this material backwards. Essentially, this is a musical reflection, or palindrome. See: `https://doi.org/10.5258/SOTON/D1387` for an audio example.

Technique 9: Phase shifting

'Phase shifting', or 'phase offsetting' is possibly the most powerful, economic tool I have employed; extracting the maximum amount of musical interest from short phrases of material via simple manipulations. *Phase music* is often considered a subgenre of minimalism though, strictly, it is not necessarily related [211]. Specifically, phase music refers to the compositional technique by gradually offsetting two simultaneous, identical ostinati so that parts lose synchronicity over time [212]. This operation results in emergent rhythmic, harmonic and timbral variation exclusively generated by the process. There could be potential distinction between phase music and *polyrhythms* (the first, parts differing in tempi, the second, contrasting metres – in this case, the same material with an additional prolongation, or subtraction to engender phasing) but I would argue that both approaches are practically congruent. Gradual offsets of a 1024th note will be perceived as subtle and timbral and will not be, pragmatically, different to true phasing. Thus if we are to conflate the two

approaches, phase shifting can be subject to quantisation: shifting by more prominent, noticeable and rhythmic durations to 'skip to the interesting parts'.

To increase the number of possible patterns, more voices can be added, however there is an upper limit to how much variation this provides. As shown in Figure 4, the maximum number of voices is contingent on the metric quotient; any number of voices larger than the value of the quotient will fail to create any new patterns through duplication. Therefore, the most musically lucrative strategy is to increase the number of beats in each sequence.

Although when participating voices are considered individually, materials have very low complexity (they are short repetitions), the instrumental *gestalt* has comparatively high complexity; with each offset, a new rhythmic pattern occurs in the combinatory image. In short, phase shifting communicates an illusion of complexity via the gestalt.

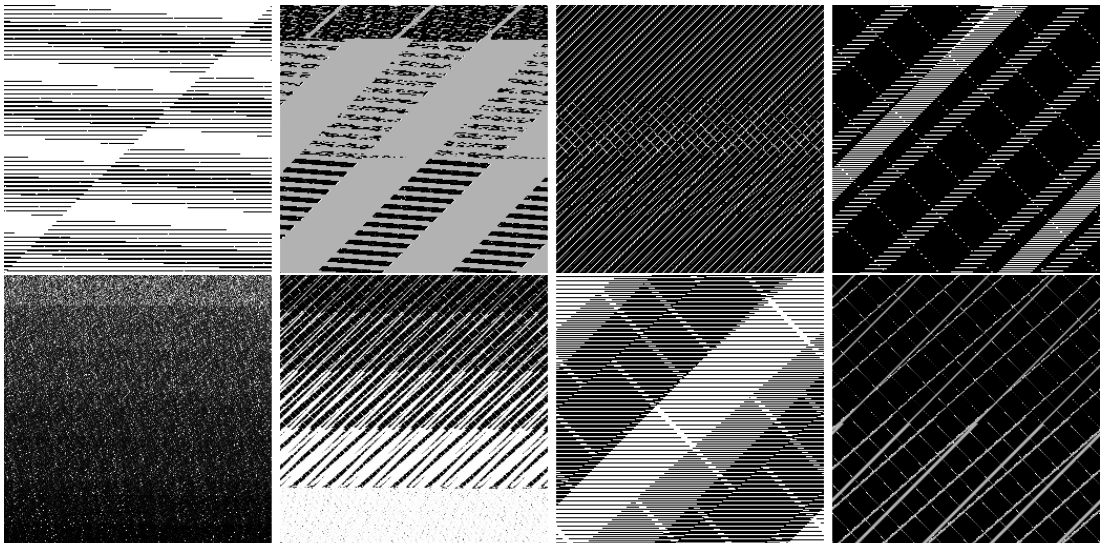


Figure 28: A selection of visualisations of short sections of the *Jupiter* μMML piece. The map is created by plotting each sample's amplitude from the minimum 0 (black) to the maximum 255 (white) from left-to-right, top-to-bottom. Each pixel represents 511 samples; the final brightness of the image is calculated by averaging the amount of waveform high events over the sampling duration. Generated at 212500Hz by the `1-bit-generator.c` program (`https://doi.org/10.5258/SOTON/D1387`).

Table 4 demonstrates the sequence of unique phase combinations possible when employing phase shifting in time signatures $\frac{e}{n}$ for c voices (or channels), where each e is offset by duration n (n is any metric denominator and e is the metric quotient). The pattern is as follows: $\begin{smallmatrix}\text{Metric Quotient} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10\\ \text{Unique Combinations} & 1 & 2 & 3 & 5 & 7 & 13 & 19 & 35 & 59 & 107\end{smallmatrix}$ and matches sequence number 'A008965' in the *The On-Line Encyclopedia of Integer Sequences* (OEIS), which can be described by the following equation[213]:

$$C(z) = \sum_{k \geq 1} \frac{\phi(k)}{k} \log \frac{1}{1 - A\left(z^k\right)} \tag{1}$$

I stumbled upon this sequence when trying to predict the number of unique patterns generated when employing phase shifting across multiple voices for a sequence of e elements. For v voices and e elements (elements are the considered the smallest possible offset in duration, or the metric denominator) how many unique patterns exist? More specifically, in a matrix of v rows and e columns, how many unique column combinations exist, regardless

e = 1	e = 2	e = 3	e = 4	e = 5	e = 6
v1 1	v1 1 2	v1 1 2 3	v1 1 2 3 4	v1 1 2 3 4 5	v1 1 2 3 4 5 6
	v2 1 2	v2 1 2 3	v2 1 2 3 4	v2 1 2 3 4 5	v2 1 2 3 4 5 6
	2 1	2 3 1	2 3 4 1	2 3 4 5 1	2 3 4 5 6 1
		v3 1 2 3	1 2 3 4	1 2 3 4 5	1 2 3 4 5 6
		2 3 1	3 4 1 2	3 4 5 1 2	3 4 5 6 1 2
		3 1 2	v3 1 2 3 4	v3 1 2 3 4 5	1 2 3 4 5 6
			2 3 4 1	2 3 4 5 1	4 5 6 1 2 3
			3 4 1 2	3 4 5 1 2	v3 1 2 3 4 5 6
			v4 1 2 3 4	1 2 3 4 5	2 3 4 5 6 1
			2 3 4 1	2 3 4 5 1	3 4 5 6 1 2
			3 4 1 2	4 5 1 2 3	1 2 3 4 5 6
			4 1 2 3	v4 1 2 3 4 5	2 3 4 5 6 1
				2 3 4 5 1	4 5 6 1 2 3
				3 4 5 1 2	1 2 3 4 5 6
				4 5 1 2 3	3 4 5 6 1 2
				v5 1 2 3 4 5	5 6 1 2 3 4
				2 3 4 5 1	v4 1 2 3 4 5 6
				3 4 5 1 2	2 3 4 5 6 1
				4 5 1 2 3	3 4 5 6 1 2
				5 1 2 3 4	4 5 6 1 2 3
					1 2 3 4 5 6
					2 3 4 5 6 1
					4 5 6 1 2 3
					6 1 2 3 4 5
					1 2 3 4 5 6
					3 4 5 6 1 2
					4 5 6 1 2 3
					6 1 2 3 4 5
					v5 1 2 3 4 5 6
					2 3 4 5 6 1
					3 4 5 6 1 2
					4 5 6 1 2 3
					5 6 1 2 3 4
					v6 1 2 3 4 5 6
					2 3 4 5 6 1
					3 4 5 6 1 2
					4 5 6 1 2 3
					5 6 1 2 3 4
					6 1 2 3 4 5

Metric Quotient (e)	1	2	3	4	5	6	7
Unique Combinations	1	2	3	5	7	13	19

Table 4: Possible unique configurations of overlapping, identical sequences in time signatures $\frac{e}{n}$ for c voices (or channels), where each e is offset by duration n (n is any metric denominator and e is the metric quotient). Voices are indicated by v and, as more voices are added (as long as v is below e), there will be additional unique variations.

of row order? Row order is inconsequential as, for the sequence C, E and G, a combination of $\binom{\text{C}}{\text{E}}_{\text{G}}$ is musically equivalent to $\binom{\text{E}}{\text{G}}_{\text{C}}$; the notes are expressed by different voices, but the configuration is identical to the listener[122]. It is obvious that, for $v > e$, there will be no additional unique patterns beyond $v = e$; there must be duplication. Furthermore, there will only ever exist one possible pattern for one voice as unique interactions are relative to contrasting voices, not relative to, or against the beat: for example, the sequence: 1 2 3 4 is the same as: 2 3 4 1, or: 3 4 1 2. This is because, when these permutations are created by offsetting a single voice alone, it will sound identical to the listener (unless the listener provides their own beat, which I consider to be a contrasting voice!), albeit with a possible crotchet pause as the sequence is 'shifted'. As for this sequence's application and utility in the compositional process, the composer only needs to know that the total possible unique patterns increases non-linearly as e gets bigger.

Although not strictly phase shifting, `paganinis-been-at-the-bins.mmml` uses the inherent metre informality of μMML (metre is indefinite and decided by the composer, not by the system) to generate interest by contrasting looped materials of different lengths in a manner very similar to that described above.

Final Thoughts

There is no doubt that there are further, perhaps less intuitive approaches, that have been omitted here. The techniques that have been explored in this section are biased to a particular form of raster composition, often found in wider chipmusic practice. Those that are employed most often are those which communicate more classical, *popular* musical tropes effectively. This type of composition is a reasonably safe method of retaining listener interest over an extended duration as it engages with a musical language in which most are familiar. If one relaxes this pre-conceived, perhaps unconscious, limitation that low-memory composition is attempting to replicate something from a different paradigm, then a new kind of music emerges. This is one that still operates in the pursuit of low B/s (and the investigation's overall premise), but also one that embraces what the medium easily creates; or rather, what readily emerges from this practice. For example, rather than using phase shifting as a feature within a wider composition (as is heard in `paganinis-been-at-the-bins.mmml`), why not use this technique as the primary compositional device (for example `jupiter.mmml`)? In this project I have made a concious choice to prominently display examples of both styles as to explore a wide gamut of compositional options available to the low-memory musician.

In focusing less on raster composition and, ultimately, further removing agency, one's predisposition to use traditional compositional objects (for example, harmony, melody and, as we'll see in the following chapter, the concept of notes and rhythm) is somewhat moderated. As such, new kinds of music are allowed to emerge.

[122] Unless played on very different instruments with differing timbres, but the assumption is that these are to be expressed by square waves — or at least the same set of instruments. Even so, the rule still holds if we concern ourselves with purely musically theoretical properties, not the resultant soundworld.

4.4 Generative Approaches

4.4.1 On The Nature Of Generative Composition

Where pre-composed material is almost entirely constructed by the composer's intentionality, in a generative piece, the composer sacrifices some agency to the process. This is not to say that the piece is truly aleatoric, it is dictated by a strictly deterministic system, only the precise results of this system may not easily, nor entirely, be prognosticated by its author. Predictability in generative music is obfuscated away from the composer by an inability to entirely determine how a set of rules will unfold [214], the extent of this indeterminacy is contingent on the creative decisions delegated to the system. An example of a 'soft' indeterminacy is demonstrated in the *Simple Oscillator* program.

```
if(fxCounter-- == 0){
  fxCounter = FX_SPEED;
  waveform--;
}
```

Listing 11: Simple square wave oscillator program (`osc.c`).

In Listing 11, the value of the *waveform* variable is not explicitly defined, but described algorithmically. The variable is continuously decremented, recurrently underflowing. Here, the procedure is obvious, little thought is required to predict the next state, yet the sequence cannot be determined without a deconstruction of the function. In a compositional environment, generative functions can be seen as compressed decelerations of their literal, raster equivalents. For example, the object `waveform[255] = {255,254,253, ... 0}` is a literal expansion of the statement `for (waveform = 255; waveform > 0; waveform--)`. In this context, functions that describe analogous raster arrays in equal to, or more bits, than their literal analogues can be considered circumlocutory.

No matter how simple and perspicuous the initial instruction set, there is a veil of incomprehension between the process and the product. The outcome is an exemplar of emergent phenomena [215, 172] plainly demonstrating how a series of very simple operations can generate complex behaviours in higher order abstractions. This raises an interesting dilemma: to whom can authorship be ascribed? We can immediately disregard the microcontroller as composer. The constituent instructions are inherently 'portable': enactable no matter the process by which they are conceived or enacted. As already discussed, 1-bit music is fundamentally platform agnostic. Furthermore, if one argues that the potential for sonification (or that, for any waveform, there exists a perfect, mathematical description) may be considered, in itself, a valid piece of music, this composition could be constructed by any entity that can perform the necessary calculations - by means of electron or by ink. The microcontroller here is unequivocally an agent in realisation, though more akin to a performer.

The logical conclusion to the problem of authorship attribution is the notion of composer as curator. In this case, the generative musician relies on a process of trial and error; a series of scripts are written, their product expanded, made accessible to human ears, then only those that cohere with the artist's value judgements are archived and presented as music. When

I come to analyse the project's generative products, I do so under the influence of *survivor bias*; I have chosen which experiments are of interest. I feel this arbitration is the primary voice of the musician in the generative process. In my mind, it operates in a paradigm similar to collecting stones at a beach: the selection is a reflection of the collector's idea of desirable.

On first inspection, a generative process seems to best satisfy the research criterion (as to whether it is possible to fit complex music in less than 1KB) as one does not need to store extra information beyond the routine. If raster, pre-composed, data is periphrastic, consuming space with superfluous literal declarations, why entertain any other strategy but generative? Many programs that algorithmically compose can be memory intensive; any neural network/machine learning implementation is largely infeasible within the spaces outlined in the project manifesto[123]. Additionally many techniques, such as the software examples of *Markov chains* and musical *sieves* in Xenakis' *Formalized Music* [217], are at least a few kilobytes in size, ignoring required libraries. Simply, even if an economical generative implementation is possible, the memory footprint for this program must be smaller than raster approaches (more straightforward sound routines total a few hundred kilobytes in size[124], outlined in 3.1, additional clever, strategic musical writing can cause small additions to make a large musical impact, see 4.3) to be considered a valid solution. As the product of simple generative processes are likely to hold less interest (and produce more homogeneous material) to the listener than that produced by humans, to be considered a valid alternative, a terse, *epigrammatic* method of algorithmic generation (no more than a few tens of bytes) must be found. It seems that a solution (at least in my own practice) is *bytebeat*.

4.4.2 Bytebeat

Formalised in September 2011, by Ville-Matias Heikkilä in a video titled *Experimental music from very short C programs*[125] [218], bytebeat is music created with no instruments nor score, but algorithmically, as a function of time [219]. Bytebeat operates at the sample domain, retaining compositional authority at the smallest level of musical time[126]. At this speed, if one were to compose in raster, huge volumes of data would be required — in the hundreds, or thousands, of kilobytes — as every sample need be stored for playback[127]. Instead, bytebeat is purely generative: short statements in code perform a series of operations on a single variable, traditionally labelled `t` for time, which is incremented endlessly; repeating on variable overflow. The benefit of this practice is that only a few tens of bytes are required in source code, in many cases, a single line of code potentially producing minutes of unique music. An example of the format is as follows:

The code in Listing 12 is short, concise and generates unique content for around a minute of music. Variable `t` (presumably 32-bit here) is incremented by the `for` loop indefinitely.

[123] As evidence, the reader may examine any library mentioned in the cited list, they are all *far* beyond a few kilobytes [216].

[124] If that; this project has used a C to AVR ASM compiler, one can create much smaller applications with assembler [131].

[125] I use 'formalised' rather than 'created' here as there has been discussion that these kind of techniques existed before Heikkilä's video, but the recent interest and subsequent categorisation most certainly dates to this event.

[126] The practice of 'sub-sample' composition (e.g. that smaller than an individual sample) is unintelligible as it does not refer to any sonified products.

[127] Essentially, crafting a wave file by hand.

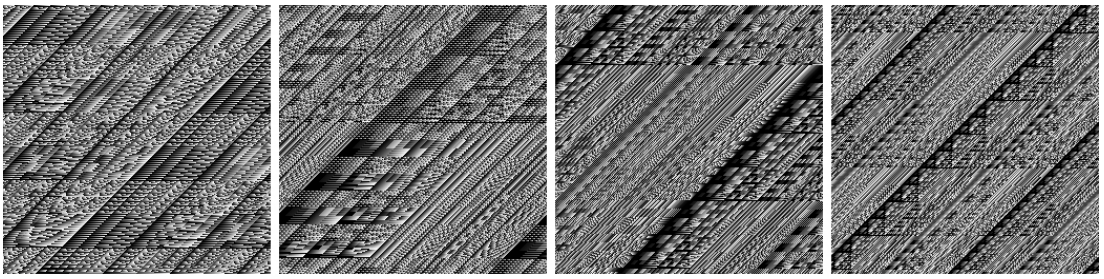


Figure 29: Product of the formula: `t*((t>>12|t>>8)&42&t>>4)-1` discovered/created by Heikkilä [218]. Images are created by plotting each sample's amplitude from the minimum 0 (black) to the maximum 255 (white) from left-to-right, bottom-to-top. Each image is 'zoomed out' by the labelled scaling factor by sampling amplitude at specified intervals. Generated at 8000Hz by the `bytebeat.c` program. See `https://doi.org/10.5258/SOTON/D1387` for an audio example.

```
int t = 0; for(t=0;;t++){
  putchar( t*((t>>12|t>>8)&42&t>>4)-1 );
}
```

Listing 12: Bytebeat piece listed in Heikkilä's seminal video demonstration.

Within this loop, a set of calculations is performed on `t`. This final result is sent to the output as raw, PCM data for playback, dictating an expressed, 8-bit, amplitudinal value. The process is then repeated for `t` now at an incrementally higher integer, providing different results with each calculation. The music generated by this process is chaotic, noisy and intricate, loosely resembling the textural landscape of chipmusic. This association is possibly due the prevalence of saw and pulse patterns in the final waveform. These are a consequence of bytebeat's particular transformations: gradually incrementing a counter is likely to engender 'ramps' of amplitude, resulting in saw-like timbres and overflowing (or underflowing) of variables will 'jump' amplitudes to their maximum or minimum states, causing pulse-like behaviour.

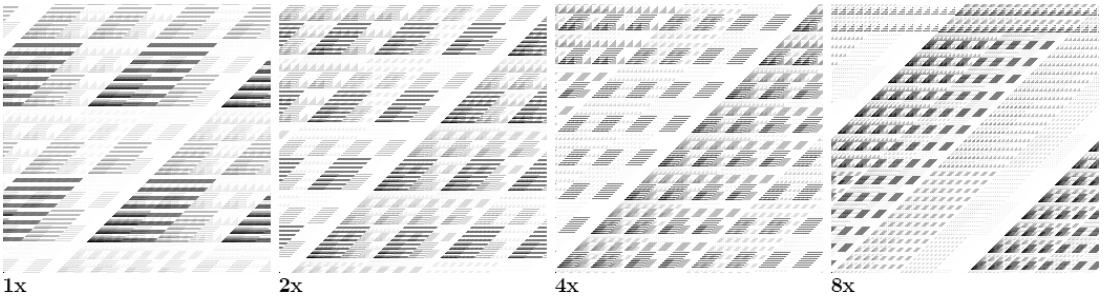


Figure 30: Product of the formula: `((t<<1)^((t<<1)+(t>>7)&t>>12))|t>>(4-(1^7&(t>>19)))|t>>7` discovered/created by Kragen [219]. Each image is 'zoomed out' by the labelled scaling factor. Generated at 8000Hz by the `bytebeat.c` program. See `https://doi.org/10.5258/SOTON/D1387` for an audio example.

Academic materials concerning bytebeat, or bytebeat application, are currently limited, however a whitepaper published in 2011 by Heikkilä formally documents the phenomenon and, alongside a detailed deconstruction on his blog [220], outlines techniques discovered collaboratively, by the wider community [221]. The paper catalogues a set of parameters that can be altered to predictably influence how emergent structures might unfold. Consequently, bytebeat composition becomes more than pure 'discovery', as Heikkilä lists each composition

in his videos, but, instead, a mixture of experimentation and deliberate composer interventions coaxing, actuating and moulding the output in predictable orientations. Bytebeat is, inarguably, elegant and beautifully simple — especially when considering its incredible memory to product ratio; seemingly a perfect solution to the research's proposed questions. Unfortunately, many of the techniques outlined in this material do not translate neatly to purely 1-bit environments.

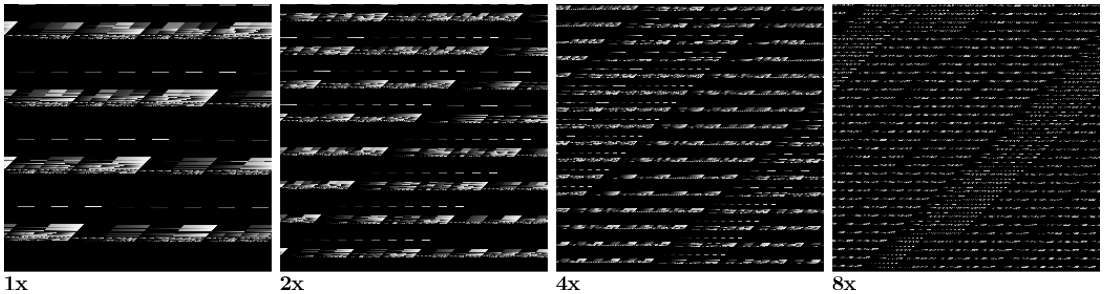


Figure 31: Product of the formula: `t*(t>>((t>>9|t>>8))&63&t>>4)`, titled: *Space Invaders VS Pong*, discovered/created by Visy [218]. Each image is 'zoomed out' by the labelled scaling factor. Generated at 8000Hz by the `bytebeat.c` program. See `https://doi.org/10.5258/SOTON/D1387` for an audio example.

4.4.3 Bitbeat

Bitbeat is a derivation of bytebeat and my solution to creating generative music with a minimal memory footprint using a 1-bit output. The core principle behind bytebeat and bitbeat algorithms is largely the same; both methods use a series of simple operations to create music from short programs, but there is a significant difference: bitbeat produces a 1-bit signal. This means that any product of the bitbeat process will conform to the same restrictions and behaviours described in sections 2, 2.2.2 and 2.2.3. Bitbeat *appears* to have 'fractal' properties, in that, over time, it is (musically) self similar. The same compositional ideas return repeatedly, often with variation. The impressive thing is that, when one considers how many samples must be created to generate a tone, bitbeat algorithms will oscillate between extreme repetition (generation of a square wave of constant pitch) to bursts of noise, pitch sweeps and complex timbral modulations. This surprising musical diversity makes it the most efficient low-memory 1-bit music solution I have explored.

Listing 13 demonstrates a simple example of a bitbeat algorithm. We can concern ourselves with the following line of material: `PORTB = (t >> PORTB | PORTB >> 1)^ 1;` which is responsible for the structure and timbre of the resultant piece. In bitbeat, the variable `PORTB` is iteratively processed, the resultant is used as the input value for the next iteration. This is distinct from typical bytebeat practice where the input to the function is a linearly incrementing `t`. One might expect that existing bytebeat formulas would be able to be transferred across to a 1-bit environment and produce comparable results, interestingly this is not usually true. The standard bytebeat approach (a series of manipulations to a single, accumulating counter (usually `t`)) only provides limited timbral interest when applied to a binary output. The main difference between bitbeat and bytebeat therefore, is that bitbeat seemingly requires more *stochasticity*. The iterative behaviour provides this, generating up to

tens of minutes of novel material[128].

```
#include <avr/io.h>

int main(void)
{
  DDRB = 0b00000001;
  for(uint32_t t = 0 ;; t++)
  {
    PORTB = (t >> PORTB | PORTB >> 1) ^ 1;
    for(uint16_t i = 0; i < 100; i++) asm("nop");
  }
}
```

Listing 13: The AVR C program code for a simple bitbeat piece. Interestingly, whilst this code continues to generate novel variations of the 'theme' for the duration of the algorithm, periodically it will pause, creating a piece not unlike Figure 17. It is baffling to me that such a simple algorithm creates such variation and complexity! An audio file of this algorithm can be generated using the `bytebeat.c` program (see: `https://doi.org/10.5258/SOTON/D1387` to download the program).

There is a (related) problem I encountered when experimenting with bytebeat: any algorithm will produce novel material for as long as there continues to be unique states of `t`. Most bytebeat is effective for only a minute or so of material. One can increase the datatype for `t`, allowing for more unique results but, ultimately, the same patterns begin to emerge repeatedly. For example, although Figure 30, produces continual variations, these are subtle and the piece's main framework — the prominent rhythmic and melodic arrangement — does not vary. If one were to expand these techniques into complete pieces (rather than sonic canapés) by adding more code, the footprint would become progressively less economical as more lines of bytebeat were added. In typical sound routines, once the core routine has been written and main memory down-payment made, subsequent raster material is likely to expand more efficiently than this initial chunk of code; especially so if observing techniques covered in Section 4.3. Additionally, if one wishes to increase the sampling rate to something higher than 8kHz, as a new state for `t` is generated every sample, bytebeat will 'burn' too quickly through samples; there must be more data per second otherwise repetitions, caused by `t` overflow, occur too rapidly.

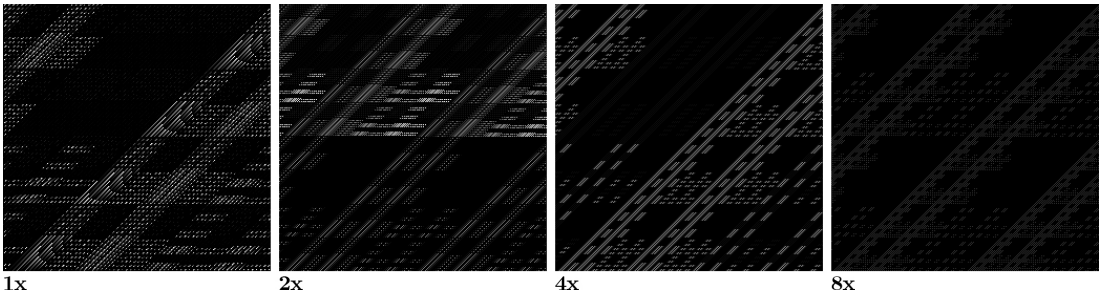


Figure 32: Product of the formula: `((t*((t>>v)&(t>>v)))&((t>>12|t>>8)&42&t>>4)-1); v-=1000;`, discovered/created by me! Uses an additional line of code to modulate variable `v`, which results in a 'phasing' effect. Each image is 'zoomed out' by the labelled scaling factor. Generated at 8000Hz by the `bytebeat.c` program. See `https://doi.org/10.5258/SOTON/D1387` for an audio example.

[128] Your mileage for any given piece may vary. Although bitbeat technically creates novel material for *hours*, I have found each piece to be interesting for only a certain amount of time. Sometimes this can be tens of minutes, other times, seconds.

Therefore, to squeeze as much content as possible into a single line, one must find a way to get even more unique material out of a the bytebeat algorithm. The initial strategy may not be in the spirit of the original premise, but further code can added to achieve this. Figure 32 is a traditional bytebeat algorithm however extended by an additional counter. The secondary variable introduces another modulation source, allowing for more complex manipulations. One can imagine this as a similar effect to phase shifting, where two, high resolution counters, decrementing by different values, will rarely repeat in synchronicity, thus producing further complexity by their contrasting cycles. This technique can be taken further to generate gradual, structural manipulations, extending interest over longer durations; a perfect solution to the 1-bit problem. This is why PORTB is used recursively; it effectively adds an additional variable alongside `t` further modulating the signal, introducing additional stochasticity. It is these additional, nested modulations that makes bitbeat interesting. Consider Listing 13 in comparison to Listing 14. Listing 14 is identical to Listing 13, however with a modification to the second `for` loop. In Listing 13, temporal resolution (sampling rate) is mediated with the inline assembler `asm("nop")` command, embedded in the `for` loop. Simply, the `nop` command 'wastes' a clock cycle. The function moves the timer away from operating against the Attiny's internal RC oscillator and into software, reducing the maximum attainable *instructions per second* (IPS). This serves to bring the generated tone into both audible and comfortable frequencies for humans and creates a buffer, allowing the routine's operational speed to be adjusted. Simply, this whole loop could be ignored if the microcontroller clock speed matched the desired sampling rate. However Listing 14 exploits this short loop by introducing a second source of modulation; time between amplitudinal events. Whilst the main melodic contour and phrasing of the piece is largely the same between the two programs, the result in Listing 14 has significantly more timbral variation; if Listing 13 was the piece's outline, Listing 14 populates this sketch with timbral 'colour'.

```
#include <avr/io.h>

int main(void)
{
  DDRB = 0b00000001;
  for(uint16_t l = 150 ;; l+=50)
  {
    for(uint16_t t = 0; t < 65535; t++)
    {
      PORTB = (t >> PORTB | PORTB >> 1) ^ 1;
      for(uint16_t i = 0; i < (l & t); i++) asm("nop");
    }
  }
}
```

Listing 14: The AVR C program code for bitbeat piece, `millipede-call-centre.c`; an expansion of Listing 13. The piece is built around a distinct rhythmic and melodic phrase that is mutated with each restatement, gradually 'decaying' in tempo and pitch.

One may also notice that, in Listing 14 a further modulation source has been added in the form of variable `l`, incrementing for each instance the primary loop concludes (`t` reaches 65535). This variable is used as an additional stochastic source in the delay `for` loop, however additionally provides a slow-evolving macro structure beyond that provided by incrementing `t`. Although the sonic results of adding further modulation sources can be, inherently, unpredictable and chaotic, one can make deliberate modifications to bitbeat algorithms,

based on the temporal categories explored in Section 3.1, and produce predictable changes. The speed at, or temporal domain, which a variable is incremented will produce changes that are perceived to be in an identical phenomenal group to the products of traditional routines.

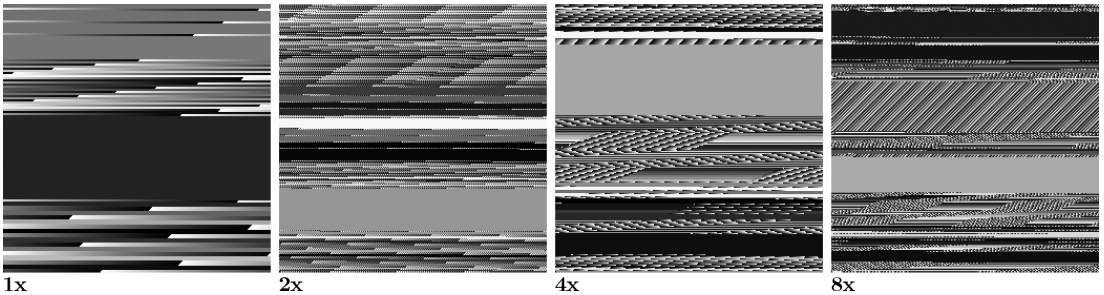


Figure 33: Visualised product of the *Tiny Djent* bitbeat piece. As the output is 1-bit, pixel brightness is calculated by averaging the total number of positive waveform samples over the image sampling duration. Generated at 4MHz by the `bytebeat.c` program.

As an additional boon, one can incorporate 1-bit techniques typically employed in raster routines. For example, at high sample rates, short, inline bitbeat formulas can be mixed via the pulse interleaving method (Section 2.2.3) to 'fill' soundscapes with multiple, parallel lines of material at varying levels of volume. Listing 15 demonstrates how this might be achieved in software. Each contiguous bitbeat algorithm updates PORTB in sequence, approximating PIM mixing. The different software channels all semantically cohere due to sharing the same delay `for` loop. As each algorithm will take varying amounts of time, thus the output of each will be expressed for different durations, some channels will be louder than others. As the compositional process is largely a random process of sifting through permutations, this is not as much of an issue as it might be for raster approaches.

```
#include <avr/io.h>

int main(void)
{
  uint16_t t,l;
  DDRB = 0b00000011;
  while(1)
  {
    for(l = 0 ;; l+=50){
      for(t = 0; t < (1024 << l); t++)
      {
        PORTB = PORTB | (PORTB >> l) ^ 1;
        PORTB = (t >> PORTB) | (PORTB >> l) ^ 1;
        PORTB = (t >> PORTB | PORTB >> 1) ^ 1;
        for(uint16_t i = 0; i < (l & (t * l)); i++) asm("nop");
      }
    }
  }
}
```

Listing 15: The AVR C program code for bitbeat piece, `infinity-soup.c`.

As, on many systems, the smallest addressable memory element size is usually one byte large [222, 223], often the bitbeat output variable is still an 8-bit number. This means that manipulations to the output will apply to all bits in a byte, even if only a single bit is sonified. If, for example, the output controls a register which affects the logic level of a

microcontroller I/O port, each bit of this register could refer to an individual 1-bit output. Consequently, multiple simultaneous, semantically related lines of material are generated synchronously, with the same code. Because bitbeat relies on a binary output, manipulating the 8-bit `PORTB` I/O register in equations gives a total of one channel per bit, eight channels in total. These channels can be either mixed in software (see 2.2.3), or simply wired to a separate output electronically[129]. This makes bitbeat more economical than bytebeat: more content is created with each algorithm and, most crucially, each sounds homogeneous.

Additionally, as bitbeat is embedded, there are only a limited set of operators and functions permissible. In bytebeat, the composer-programmer's intention is to reduce the number of characters in the source code, in bitbeat it is to reduce the compiled size. This may not seem significant, however this restriction makes previously trivial operations costly to implement. For example, the modulo function in AVR C requires an additional kilobyte of program space and even the multiplication and divisor operators will add an additional fifty (or so) bytes to the compiled size [147]. Therefore, to keep filesize to a minimum, one must rely on those operations which are compiled smallest; in AVR C these are, primarily: addition, subtraction and bitwise operations.

Bitbeat might be somewhat unpalatable to the average listener; it does not adhere to standardised tunings and rarely produces consonant intervals, but it is undeniably musical. Pieces often have a regular rhythm and are subdivided into common time (like Listing 13). I consider this to be somewhat of a victory, it does seem possible to produce something at least resembling music in very low memory environments. The smallest, coherent bitbeat algorithm in this project (Listing 13) continues to extrapolate on a core idea for tens of minutes (despite lengthy pauses between seconds) and sizes just 104 bytes.

[129]The reader should be aware that fewer channels may be generated if there is an destructive bitwise shift left (or right) before expressing outputs.

5 Conclusion

Whilst it is certainly possible to create music within a single kilobyte of Attiny microcontroller flash memory, the limit on how engaging this might be is dependent on the method of generation. Whilst μMML is undoubtedly not the most efficient raster solution possible, it is difficult to imagine how one might fit a piece like 4000AD into 1KB. Even if the initial routine was refactored, or rewritten in AVR assembler, the data array would remain the same size. To mitigate this, one might delegate certain compositional elements to simple in-line, user addressable functions (such as automatic arpeggios, or melody generators, for example) however, for the composer to communicate their ideas unabridged (lossless), there will remain as many symbols as there are events required to replicate the idea. The number of required symbols may be reduced using data compression methods, but this expands the routine size in order to make savings in the song data. This is only sensible if one will see a return on this memory down-payment by means of reduced expansion in future, which is unlikely true in 1KB. Essentially, what I am describing is a form of the *space–time tradeoff* (or time-memory tradeoff) [224, 147]. Although one could consequently consider this problem as the concern of computer science, perhaps the most interesting thing about this endeavour is that, as composition is creative, there is no pre-existing object to compress; one may build *within* the data boundaries. The composer can completely yield to the limitations and expand where the system is most forgiving. They may select only those creative choices which work best, thus producing artefacts that seemingly belie the constraints via intimate knowledge of the system itself[130]. Simply, as a creative, one can change their intention to work with the system.

To demonstrate, if we were to imagine the data limits of low-memory composition as a limited pool of hexadecimal characters instead (A, B, C, D, E and F, including punctuation), the approach produces an equally effective result:

Before ‘compression’:

```
a run-down restaurant

a faded, defaced cafe
```

After ‘compression’ (removing all letters that are not hexadecimal characters):

```
a -d eaa

a faded, defaced cafe
```

Whilst the second sentence is not semantically identical to the first, and adds a few creative substitutions (for example, ‘restaurant’ and ‘cafe’ are not directly synonymous), it forces alternate solutions and *seems* to contradict the six character limitation; it subverts expectations as to what is possible within the framework. In short, if the composer does not endeavour to create generalised solutions (as the computer scientist or mathematician might),

[130] Effectuated in a manner similar to technique seven in Section 4.3.

the composer has liberty to create smaller, more perspicacious, more idiosyncratic products. This is why low-memory composition is worth exploring, it both challenges and changes compositional models in a new way — not like serialism, new complexity or spectralism where traditional, theoretical models are challenged — but *semiotically*, where the traditional management of musical instructions are challenged. I would go as far as to suggest as low memory composition can be an exaggeration and amplification of instrumental composition. As the maximum permitted filesize decreases, the various creative concessions one might occasionally expect to negotiate instead consume the composer's decisions, to such an extent that effective composition becomes a puzzle.

So, if we are to evaluate the success of this project by the 'bytes-per-second' criteria presented in Section 4, for each of the pieces in the portfolio, we must ascertain which has the smallest B/s — indicative of the most effective low-memory compositional technique. I find it difficult to decide which of my devised solutions and memory optimisation techniques is the winner, bitbeat or μMML, as each has individualistic strengths and, within those two methods, there are many approaches of varying effectiveness. Table 5 lists each of the project's μMML pieces, ordered by B/s. Bitbeat pieces have not been included as they are somewhat problematic to evaluate using the same criteria as the μMML pieces, with particular respect to maintaining musical interest. Technically one can argue that each bitbeat algorithm has the potential to create unique material for hours (perhaps even days), however these variations can be either quite subtle or, more often, simply uninteresting after a few minutes, as the emergent arrangements do not deviate too far from a few core musical 'ideas'. As I have no established model by which I can objectively evaluate the musical complexity[131], and subsequently interest, of any given piece (or taken an average complexity over a period time), these are excluded from the analysis.

As is evident from Table 5, the only μMML pieces that answer the question as to whether is it actually possible to create complex, engaging music in less than a kilobyte of program space, are the *Piano Phase* (`piano-phase.mmml`) and Bach (`prelude.mmml`) transcriptions, along with the *Jupiter* suite (`jupiter.mmml`). However, this is only if one excludes the μMML routine, which sizes 1458 bytes in total. Although this means that *any* piece made using μMML will exceed the research's memory limitations, with significant refactoring, removing superfluous functions and converting `mmml.c` to AVR assembler, it may be possible to squeeze some of these pieces into a single kilobyte. Interestingly, two of these pieces use phase shifting as their primary compositional technique; it is clearly the most efficient approach to raster composition explored. In fact, file sizes are so small that they rival that of bitbeat (shown in Table 6). I think that, if one considers timbral complexity, compositional interest and B/s, `jupiter.mmml` is by far the most successful composition. Although perhaps not as conventionally interesting as `4000ad.mmml`, or `goose-communcations.mmml`, it is more engaging than the bitbeat compositions over a longer period of time. The Bach transcription falls below a kilobyte for two reasons: firstly, timbral variation has been sacrificed for memory savings and secondly, I deliberately selected a piece that had a significant amount of repetition and would work well with the μMML language (perhaps another example of 'colouring within the lines').

[131] Currently anyway; I would like to develop something to allow more objective evaluations in future. Beyond musicological applications, this could also enable more sophisticated, computational methods of low-memory composition.

Without compiled `mmml.c` routine size:

Title	Size (Bytes)	Duration (Seconds)	Bytes Per Second (B/s)
Piano Phase	85	143	0.59
Jupiter	646	676	0.96
Bach Prelude	661	120	5.51
4000AD	6495	489	13.28
Greatest Hat	1590	102	15.59
Paganini's Been At The Bins	3410	209	16.32
Sunglasses Snake	3246	191	16.99
Puppy Slug	2016	118	17.08
Til There Was You	2338	130	17.98
Goose Communications	5377	281	19.14
Fly Me To The Moon	2029	105	19.32
Shrub Club	3291	168	19.59
Goblin Shark	3574	174	20.54

With compiled `mmml.c` routine size:

Title	Size (Bytes)	Duration (Seconds)	Bytes Per Second (B/s)
Jupiter	2104	676	3.11
Piano Phase	1543	143	10.79
4000AD	7953	489	16.26
Bach Prelude	2119	120	17.66
Paganini's Been At The Bins	4868	209	23.29
Goose Communications	6853	281	24.32
Sunglasses Snake	4704	191	24.63
Shrub Club	4749	168	28.27
Goblin Shark	5032	174	28.92
Til There Was You	3796	130	29.2
Puppy Slug	3474	118	29.44
Greatest Hat	3048	102	29.88
Fly Me To The Moon	3487	105	33.21

Table 5: Two tables comparing the B/s of different pieces created using the `mmml.c` program. The first lists piece sizes without the routine included, the second lists the compiled size of both routine and piece. Compiled routine sizes will be subject to the version of compiler but, as the datasize is fixed whatever the platform of compilation, the B/s without the routine is global.

Whilst `4000AD.mmml`, `paganinis-been-at-the-bins.mmml` and `goose-communications.mmml` do not meet the originally proposed 1KB total limit, I feel these are the most classically interesting pieces I have produced for microcontroller. These may not be as conceptually intriguing as bitbeat, nor as efficient as `jupiter.mmml`, however they demonstrate sophisticated compositional compressional practice utilising (and inspired by) classical chiptune instrumental writing. These pieces play with the techniques discussed in Section 4 to push the 1-bit palette, the Attiny85 and explore the limitations of the μMML language. In fact, there are no existing low-powered, embedded 1-bit music suites in existence that take the same counterpoint focused approach. I have a personal affection for this kind of practice as I find it is the most enjoyable and rewarding way to write low memory music, even if the works do not contain the most progressive methodologies explored. Still, `4000AD.mmml` has an impressive B/s; the third most efficient when the routine size is considered (beating the heavily compressed `bach-prelude.mmml`) and suggests potential evolutions and extrapolations of the practice (see below). This kind of composition requires a larger file size before memory savings begin to be noticeable.

This research may be of interest to ludomusicologists, as the compositional motivations of low-memory video game composers working in the 1980's and 1990's are similar to my own, but have often gone undocumented. The process of building music routines for Attiny documented herein is analogous to designing routines for almost any electronic computational system, additionally, any platforms employing 1-bit sonics *must* use those techniques

Title	Data Size (Bytes)
Millipede Call Centre	116
Ghost Pony	142
Modem Exorcism Pt.1	156
Small Ahh	180
Upstairs Neighbours	184
Snooping	186
Modem Exorcism Pt.2	190
Typewriter Tantrum	190
Helicopter Mating Season	196
The Squeak Squad	202
Infinity Soup	204
Inkjet House Party	204
Sludge Bugs	210
Howl Owl	218
Fax Attack	230
Ghoul School	238
Tiny Djent	354

Table 6: A table comparing the B/s of different pieces created using bitbeat. Like Table 5, compiled sizes will be subject to the version of compiler, but will still stay largely consistent between toolchains.

described in sections 2.2, 2.2.2 and 2.2.3 to some extent. Moreover, the ideas presented in this document provide a novel perspective on the practice of low memory composition, bolstering a sparsely populated academic bibliography on the subject. Additionally, I feel this an esoteric look at compositional process in general.

Although the primary research of this investigation has been highly empirical, it is, at its foundation, personal experimentation. Such an approach may undermine the validity of the project as the criterion of success will be ultimately derived, and subsequently evaluated, entirely subjectively. The problem is that, if the criteria is to be minimal program size with maximal piece duration, the most successful low-memory composition are surely a single note stretched over an infinite duration[132]. As most would not consider this to be interesting, there must be something inherently subjective in the evaluation of successful parameters in music — or at least the solution is far more nuanced. To expand on, and synthesise the ideas explored in, Section 4.3 and μMML, if one were able to achieve an objective measure of compositional success, perhaps those low-memory compositional techniques outlined in Section 4 might be automated and implemented in software. Perhaps a program can be built that obeys those strategies for reducing compositional footprint explored in Section 4.3? The proposed solution would be a lossy, 'content-aware' compositional system of compression. Some of the strategies would certainly be easier to implement than others; *Technique 6: Colour Within The Lines* might be effectuated by simply 'quantising' a rhythm, however others would be much more nuanced — and far more interesting! For example, in accordance with *Technique 2: Recycling And Re-Purposing* any two musical materials that are similar in functionality, yet written slightly differently, might be substituted for each other. This could a simple matter of identifying a few different notes between two passages (for example `c8 e8 g8 c8 e8 g8` and `c8 g8 e8 c8 g8 e8`) and replacing one instance with the other, or something more complex, where the program must understand the harmonic role of two (or multiple) materials and substitute based on compositional function (for example `c8 e8 g8 c8 e8 g8` and `c4 e4 g4`). Of course, this could change a vital harmonic relationship between other

[132] A couple of bytes of assembler could achieve this. The infinite duration would require the hardware to resist entropy and the ultimate heat death of the universe, though often batteries do not last this long.

voices and deform the original piece in the process, but, depending on how much the material is employed, this might be a beneficial saving[133].

Bitbeat is the primary area that would benefit from ongoing exploration, with a number of interesting avenues that might be worth investigating. Firstly, as 'panning' for viable bitbeat algorithms via C is time intensive, it seems feasible that one might devise a programmatic system to determine any given algorithm's success. This is reliant on an absolute criterion for musical success. One might use *reinforcement learning*: evaluate an output waveform's complexity[134] over a given period of time and assign a complexity value for that period. Ideally the segment would sit within the 'sweet spot' of a Crutchfield complexity curve, or Vitz's inverted U function [176, 172]. The listener might enjoy short periods of very high complexity (bursts of percussive noise), or periods of very low complexity (a single, repeating wavecycle), but the average complexity over a 'macro' duration (which might be an average of a series of these small periods of time) should be not too complex, or too simple. An interpreted bitbeat language, where algorithms can be manipulated programmatically, would allow for mutation and self-modification of algorithms and the clear definitions of complexity would provide an objective parameter of success — without dictating the actual musical content. A series of machines dedicated to finding musical solutions might discover some exiting possibilities — perhaps, with a strict, decreasing word limit, one could discover the shortest, most interesting musical sequences possible.

Bitbeat is most exciting when one approaches the process in reverse: there exists a solution for any bitbeat recording that can be expressed in just a few lines of code. Why are some bitstreams reducible in this way, but others seemingly are not? Thus, for a bitstream n bits in length, is there a corresponding, generative algorithm of a few bytes? If not, could there be a sequence that creates an *approximate* (which there will be infinitely many more)? How might one find this approximate? This reminds me of the Library Of Babel [226]; if there was an algorithm that could create every permutation of a bitstream of n bits, could one simply calculate where a set of desired patterns appear then store pointers to those locations, or is there some 'law' that forbids this? Again, perhaps the goal of the composer could help defy this problem? One might find pieces within the function that *can* be easily addressed? Furthermore, could the bitbeat and raster routines be synthesised? Perhaps bitbeat routines could 'fill' gaps between raster materials, or perhaps a function, accessible to the composer-programmer via a command, could allow sequencing of inline bitbeat algorithms. These bitbeat algorithms could have their own interpreted language and be stored in a lookup table — somewhat similar to μMML's macro function.

Although 1-bit composition continues, and chiptune itself will certainly continue to be a compositional curiosity for some time, the pursuit of composing sophisticated music in small spaces to explore the emergent techniques is certainly diminishing. Submitted as part of this project is all the software I have written to create 1-bit, low memory music. The reader has everything required to recreate my music from first principles, or create new. Additionally, this commentary aligns the software toolchain with the sonic theory, the

[133] Having recently played with Google's *Parsey McParseface* [225], it might be valuable to exploring whether a musical equivalent of Google's *SyntaxNet* (the open-source neural network framework with which Parsey McParseface has been trained) would be possible for music, instead of language. It could prove useful in the automation of low-memory music: compressing MML, MIDI or musical score (symbolic formats) using intelligent, compositional means.

[134] Perhaps using Shannon entropy?

required instrumental techniques and effective compositional approaches. Your compositional voice will, most likely, be significantly different to mine and, if you are new to low-memory composition (or new to composition in general) you will certainly have something unique to say using the medium. I do hope that this project has intrigued and encouraged further exploration of this world; perhaps it has inspired to write more 1-bit music, or develop new low-memory platforms. In a world where prohibitively expensive contemporary sample libraries and digital audio workstations significantly contribute to professional success, accessible, low-cost and open source alternatives remain a relevant antithesis and esoteric antidote. So, experiment with the included code, expand the functionality, open a blank `.mmml` document and start composing!

6 Appendices

6.1 4000ad.mmml

```
%====================================================================%
% TITLE      : 4000AD
% COMPOSER   : Blake 'PROTODOME' Troise
% PROGRAMMER : Blake 'PROTODOME' Troise
% DATE       : 14th June 2018
% NOTES      : Computer music of the far future... 8 minutes of
%              gratuitous 1-bit wankery.
%====================================================================%

%~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL A %~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% stutter intro (4/8)
r2 [5 m1 ]

% Cmin7 arp intro (3/8), drum solo (4/4)
[51 m4 ]

% C#min7 arp intro (7/8)
[17 m9 ]
r8.

% Cmin7 bass groove (4/4), F#13 bass groove (4/4)
[21 m4 ] r8 [14 m9 ]

% transition A (4/4)
o4 v5 r64c#32.<<r64c#32.>r64c#32.
v6 o3 b16a16f+16e32f+32c+16f+16c32e32<b16a16f64f+32.>
[2 e64c+64<b64> ]    v4 [2 e64c+64<b64> ]
v6 [2 f64d64c64 ]    v4 [2 f64d64c64 ]
v6 [2 f+64d+64c+64 ] v4 [2 f+64d+64c+64 ]
v6 [2 g64e64d64 ]    v4 [2 g64e64d64 ]
v6 [2 g+64f64d+64 ]  v4 [2 g+64f64d+64 ]
v6 [2 a64f+64e64 ]   v4 a64f+64
[4 m1 ]

% solo A (4/4)
r8v6>a+64>d64c8&c32 [3 c+64c64<b64>c64 ] d+64
f+64g64d+64c64<a+64>c64<g64a+64g64f64f+64g64d+64c64<a+64g64f64g64d+64
f64d+64f64a64>c32f32a+32>d32
f32d+32c32<a+32>d+32c32<a+32g32>c32<a+32g32f32a+32g32f32d+32g32f32d+32
c32f32d+32c32<a+32>d+32c32<a+32g32>c32<a+32g32d+32
c8.&c32 [4 c+64c64<b64>c64 ] v4 <b64
a+64a64g+64g64f+64f64e64r8r32 v6 d+16f16g16a+16
>d+16 v4 d+16 v6 c16g32g+32g16 v4 g16 v6 a+16 v4 a+16 v6 g16>d8 v4 d16 v6 d+16e32
f32c16<g16
f8 v4 f16 v6 g8.&g32 [4 g+64g64f+64g64 ]
r32a+16b32r32>c16d+32.r64f16
[4 f+64g32.>c32r32< ] f32f+64
g32.>c32<f32f+64g32.>c32 [4 <f64f+64g64>c64 ]
d+32c32<a+32>c32<g32f+32f32d+32f32d+32c32d+32f+32d+32c16d+32f32d+16g16
d+16f32f+32d+32c32<a+16.>c32
v4 <b64a+64a64g+64g64f+64f64e64r2r8 v6 g+64a64a+64b64>c8.
g+16.a64a+8a64g+8.&g+32. [2 a64g+64g64g+64 ] a64>c+16r8.<b16r8
a+16g+16a+16f+16d+16c+32d+32<b16>c+16r8<f+16r2
r16b16r16
>d+16e16f+16a+16g+16c+16d+16f+16>d+8&
d+32 v4 d+16. v6 f32f+8
r32c+8. v4 c+16.r8r32. v6 <f32.f+32a+32>c+16
f32.r64f+16r64f16d+32.c+32.r32c32.<b32.r32>d+32.<a+16r64g+32r64b32>c+16.
<a+16f+16g+32a32a+16
g+16f+16d+16g+16f+16d+16c+32r32f+16f64 v4 e64d+64d64c+64c64r2
r32
v6 <c+16d+16f32r32f+16g+16a+16b16>c32r32c+16d+16f+16g+32r32a+16>c+16

% transition B (7/4)
m21

% four-to-the-floor A (4/4)
```

```
r64 [3 m27 m28 ] m27

% transition C (4/4)
o4 v5 [3 e64<g64b64>c64 ] e64<g64b64r8> [4 c+64f64<g+64>c64 ] r8 [4 d64f+64<a64>c+64 ] r8 [4 d+64
     g64<a+64>d64 ] r16
v6 >d+64<f+64d+64<b64>g+64b64>d+64<f+64r16>e64<g64e64c64a64>c64e64<g64r16>f64<g+64f64c+64b64>c+64
     f64<g+64f64c+64b64>c+64

% solo B (4/4)
o4 r64 [2 m29 r1 r1 ]
v6 o5 r32.f+64g32.c16<a+16a16f+64g32.f16g16d+32.d64c16<a+16>c32<a+32g16
e64f32.d+16c16
a32a+32>c16<g32r32>a64<a32.>f64<f32.>f+64<f+32.>g64<g32.>d+64<d+32.>c64
<c32.>d64<d32.a+64<a+32.>g64<g32.>d+64<d+32.>e64<e32.>f64<f32.>d+64<d+32.
c16r2r8>>d+16f16g16a+16b32>c16
v4 <b64a+64r8 v6 >f+16r8f+32g32a+32g32a+32f+32f32d+32c16f16d+16c16<a+16
b64>c32.
v5 <b64a+64r8.r32 v6 <c32f32d32g32d+32a+32f32>c32<f+32>d32<g32>d+32<a+32
>f32g32a32a+32>c32d32d+32f32g32a+32a32
f32d+32f32g32c32<a+32>d32d+32c32<a+32a32g+32g32f+32g32f32d+32d32c32<a+32
>d32c32<g32a+32a32f32f+32g32d+32d32<a+32>d+32
d8&d32d+32e32f4&f32 [4 f+64f64e64f64 ] d+16d16c16f16
d+16d16r16c8c+64c64<b64>c64d16d+16r16a+32g32a32a+32>c32c+32d16d+16f16
g16
g+32a+32g+32a+32g+4&g+32a32a+8.&a+32a32g+32g32r8>e64f64f+16.
e8<b16>d+16<g+16a+32b32f+16g+16d+16f+16c+16d+16<b16a+16f+16d+16
<b8.g+32b32>c+8.<a+32>d+32e8&e32b32f+32g32g+8>d32d+32g+32f+32
<b16g16f+16e16d+16f+16d16c+16<b16>f+16b16>d+16f+16a+32b32>d+16f+16
r16d+32d32c+16r16<b16r16>g32g+32e32d+32d32c+32d+32<b32>c+32<g+32f+32
d+32<b32>c+32<g+32a+32>d+32f+32g+32a+32
b16>c+16d+32r32f+16c+16<b16>c+16<g+16b16f32f+8&f+32>c+32d+8.&d+32
d64c+64c64<b64r16<c+16d+16f32r32f+16g+16a+16b16>c32r32c+16d+16f+16g+32
r32a+16>c+16

% transition B (7/4)
m21

% four-to-the-floor A (4/4)
r64 [3 m27 m28 ] m27
r2.r8.r32.

% four-to-the-floor B (4/4)
[3 m33 m34 ] m33 r1

% bass solo (4/4)
[8 m33 m34 ]

% stutter transition (4/8)
[4 m1 ] o4 v3 [16 r64 c32. ]

% movement II intro (4/4)
o4 [80 v3 e32 v2 e32 v3 c32 v2 c32 v3 d32 v2 d32< v3 g16 ]

% movement II main (4/4)
[2 [2 [2 r64 m45 r4.r32. ] [2 r64 m46 r4.r32. ] ] m42 r64 m46 r4.r32. m42 r64 m46 r4.r32. m43 m44
     ]

% movement II solo (4/4)
o5 v6 r2f64f+64g64a64g8.[2 g+64g64f+64g64 ] f+16f16
e16<b16>c16d32e32<a16f16g16g+16a16>c16<g16f16e32f32c16d16e16
c32d8&d32 v5 d8. v6 g16r16g8r16 v4 g8r16 v3 g8
r2 v6 e32g32a32a+32b16g16e16r16a+16f+16
r4>c32<a32f32c32d16a16f16>c16<a16>d16d+16e16c16<a16
f32g16. v5 g8 v6 f8 v5 f8 v6 d32e8&e32r16g8 v4 g8
v6 d8.e16r8 v5 e16r2r16
v6 a32g32f32d32c32<a+32g32e32f32d32<a+32>c32d32e32f32d32f32g32a+32>d32
f32g32a+32a32g+32g32>c32<a+32b32>c32d32e32
f16e64d64c64<b64<f16a16>e16c16<a16>c16g32a32e16c16g16b32>c32d16e16f16
e8c16<a16f32f+32g32e32d32c32<b32>c32<a32f32g32e32f32f+32g32a32b32>c32
d32e32f32g32a32a+32
b16>c16d16f32g16.c16<b16>c16<a16b16g+16a16b4.
>c8.c+64c64<b64>c64<g16d16<b32>c+16.<a8>d8e8
f4e8c16 v5 c16 v4 c16r16 v6 <a16 v5 a16 v4 a16r16 v6 >c4
v5 c8 v4 c8 v3 c8 v6 d8 v5 d8 v4 d8 v3 d8

% transition D
m51
o3 v6 [4 f+64a+64d+64 ] r16
```

```
[4 g64b64e64 ] r16
[4 >c64<f64g+64 ]r16
[4 f+64a64>c+64< ] r16 o2
[3 a+64>d64<g64 ] r32.
[3 b64>d+64<g+64 ] r32.
[3 >c64e64<a64 ] r32.
[3 >c+64f64<a+64 ] r32.
[3 >d64f+64<b64 ] r32.>
[3 d+64g64c64 ] r32.
[18 e64g+64c+64 ] e64g+64
r8 [19 f64a64d64 ] r8.r32.

% solo C (4/4)
o3 v6 >a+64b64>c32c+32d32c32<a+32>c4&c16c+64c64<b64>c64c+64c64<b64>c64d+32
[5 c32d+32 ] [4 c32f32 ] [3 c32f+32 ] c32g32c32d+32f32d+32c32<a+32
>c32<g32a+32g32f32f+32f32d+32c32d+32<a+32
b32>c32d+32f32f+32g32a+32g32a+32f+32f32d+32c16.f32d+16.c32<a+16.r32a16.
r32g16c32d+16.
[2 e64d+64d64d+64 ] f8& [2 f64f+64f64e64 ] f+16g16a+16
>c16d+32f32g32a32a+16a16g+16
g4 v5 g8 v4 g16. v6 g+32a4 v5 a8 v4 a8
v6 a+16>c16d16d+16f16g16a16g+16g16e16f16d32d+32c32<b32a+16a16f16
g8.&g32d64<a64g8.&g32e64<b64g8.&g32f64c64<g4
r8>g16a16a+16>c16d16d+16f32g16a16a+32>c32d32d+16f16g16g+32a32
a+2.b8. v5 a+8
r16b8r16 v4 a+8r16b8r16 v3 a+8r16b8
r16 v2 a+8r16b8r8 v6 g+16a+16f+16d+16c+16f+16<b16f+16
a+4 v5 a+4 v3 a+8 v6 >>d+16f+16d+32e32d+32d32c+16<b16
>c+2&c+8 v5 c+4.
v4 c+4. v3 c+4. v2 c+4
r1

% main theme reprise (4/4, 7/8)
[2 m42 v4 r64 [3 o4c32f32g+32>c32e32g32 ] v5 o4c32f32 v6 g+32>c32e32g64 ] m43

% final stretch (4/4)
m51 m54
m51 o5 v6 d+16d16<a+16g16f16d+16d16<a+16g16f16d+16d16<a+16g16f16d+16
m51 m54
m51 o5 e16d16<a16g16f16e16d16<a16g16f16e16d16<a16g16f16e16
m51 m54
m51

% ending stutter (4/8)
[4 m1 ]
o4 v5 c1 c1 v4 c1 t63 v3 c1 t69 v2 c1 t74 c1 c1

%~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL B %~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% stutter intro (4/8)
[6 m2 ]

% Cmin7 arp intro (3/8), drum solo (4/4)
[51 m5 ]

% C#min7 arp intro (7/8)
[17 m10 ]
m11

% Cmin7 bass groove (4/4), F#13 bass groove (4/4)
[21 m5 ] r8 [14 m10 ]

% transition B (4/4)
o4 v4 c#16e16>c#16
o3 r2r8 v5 [2 e32<f+32b32> ] [2 f32<g32>c32 ]
[2 f+32<g+32>c+32 ] [2 g32<a32>d32 ] [2 g+32<a+32>d+32 ]
a32<b32>e32a32
[4 m2 ]

% solo A (4/4)
[21 m5 ] r8 [16 m10 ]

% transition B (7/4)
m22

% four-to-the-floor A (4/4)
```

```
[3 m27 m28 ] m27

% transition C (4/4)
v6 o3 e64g64>c64e64g64b64>e64g32e64<b64g64e64c64<g64e64r8f64g+64>c+64f64
g+64>c64f64g+32f64c64<g+64f64c+64<g+64f64r8f+64a64>d64f+64a64>c+64f+64
a32f+64c+64<a64f+64d64<a64f+64
r8g64a+64>d+64g64a+64>d64g64a+32g64d64<a+64g64d+64<a+64g64r16>f+16>d+16
r16<g16>e16r16<b32>d+32<g32b32>d+32<g32

% solo B (4/4)
[8 o3 m29 ]
[3 m30 ] m31

% transition B (7/4)
m22

% four-to-the-floor A (4/4)
[3 m27 m28 ] m27
r1

% four-to-the-floor B (4/4)
[7 m35 ] r1

% bass solo (4/4)
[16 m35 ]

% stutter transition (4/8)
[4 m2 ] o3 [16 v3 f32>f64< v2 f64 ]

% movement II intro (4/4)
[5 [42 v3 o4 e64<d64g64 ] r32 [42 v3 o4 f64<d64g+64 ] r32 ]

% movement II main (4/4)
[2 [4 [2 [3 o4 v6 d64<g64e64 v5 >d64<g64e64 v4 >d64<g64e64 v3 >d64<g64e64 ] r16 m41 ] [2 [3 o4 v6
     d64<g+64f64 v5 >d64<g+64f64 v4 >d64<g+64f64 v3 >d64<g+64f64 ] r16 m41 ] ]
[2 [3 o4 v6 d64<f+64a64 v5 >d64<f+64a64 v4 >d64<f+64a64 v3 >d64<f+64a64 ] r16 m41 ] [2 o4 v6 c64<
     d64g+64 ] o4 v5 c64<d64g+64 v4 [39 o4 c64<d64g+64 ] r32 ]

% movement II solo (4/4)
m48 o3
[2 m49 r4 ]
m50 r4
[2 v6 b64f64d64 v5 b64f64d64 v4 b64f64d64 v3 [5 b64f64d64 ] ] r4
m49 r4
o4 v6 c64<a+64g64> v5 [19 c64<a+64g64> ] r16
m48
m49 o3
o4 v6 c64<g64e64> v5 [14 c64<g64e64> ] r32.
v6 c+64<g64e64> v5 [9 c+64<g64e64> ] r32
o3 m50 r8
v5 f4. v4 f8 v3 f8
v5 f4 v4 f8 v3 f8

% transition D
[5 m52 ] r16
o3 v6c+8.r16d8.r16d+8.r16e8.<a+64b64>c64c+64
d32g32a+32>d32r16<d+32g+32b32>d+32r16<e32a32>c32e32r16<f32a+32>c+32f32
r16<f+32b32>d32f+32r16<g32>c32d+32g32r16
[5 <c+32e32g+32>c+32e32g+32 ] r16
[5 <d32f32a32>d32f32a32 ] r8.

% solo C (4/4)
[4 o3 m29 ]
[3 m30 ] m31

% main theme reprise (4/4, 7/4)
[2 [11 o4 v4 c64 v5 c64 v4 e64 v5 e64 v4 g64 v5 g64> v4 c64 v5 c64 v4 e64 v5 e64 v4 g64 v5 g64 ]
[9 o4 c32f32g+32>c32e32g32 ] ]
[21 o4 d64f+64a64>c64f+64a64 ] r32

% final stretch (4/4)
[11 m52 ]
[10 o4 v6 f128< v5 f64.> v6 g128< v5 g64.> v6 a+128< v5 a+64. o5 v6 f128< v5 f64.> v6 g128< v5 g64
     .> v6 a+128< v5 a+64. ]
[11 m52 ]
[10 o4 v6 f128< v5 f64.> v6 g128< v5 g64.> v6 a128< v5 a64. o5 v6 f128< v5 f64.> v6 g128< v5 g64.>
      v6 a128< v5 a64. ]
[11 m52 ] o3 d+32f32
```

```
[21 o3 f64a64>d64 ] o3 f64

% ending stutter
[4 m2 ] o3
v4 [32 d32>e64<e64 ]
v3 [16 d32>e64<e64 ]
v2 [16 d32>e64<e64 ]
v1 [32 d32>e64<e64 ]
r1

%~~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL C %~~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% stutter intro (4/8)
r1 [8 m3 ]

% Cmin7 arp intro (3/8), drum solo (4/4)
[51 o3 v3 g32> v4 g32< v3 d+32> v4 d+32< v3 f32> v4 f32< v3 c32> v4 c32 v3 c32> v4 c32<< v3 a+32>
     v4 a+32 ]

% C#min7 arp intro (7/8)
[17 o3 v3 e32> v4 e32< v3 f+32> v4 f+32< v3 c+32> v4 c+32< v3 e32> v4 e32 v3 c+32> v4 c+32<< v3
     b32> v4 b32 v3 g+32> v4 g+32 ]
m12

% Cmin7 bass groove (4/4)
m14 m15 m14 m16

% F#13 bass groove (4/4)
m17 m18 m17 m19 m17 m18

% transition A (4/4)
v6 o1 f+16>>f+32r32<<f+16r2.f+8
>f+32r32<g8>g16<g+8>g+32r32<a16>a32r32<a32r32a+32.r64d32.r64>c+32r32<b16d+16
>c4 v5 c8 v4 c8 v3 c8 v2 c8 [5 m3 ]

% solo A (4/4)
m14 m15 m14 m16
m17 m18 m17 m19 m17 m18 m17

% transition B (7/4)
m23

% four-to-the-floor A (4/4)
[14 m26 ]

% transition C (4/4)
v6 o1 c8.>c16r16<c32r32c+8.>c+16<f+16g32r32d8.>d16<
g32a32g32f32d+8.a+16r16e8>c32r32<f8>d+32e32<g16.>g+64g64f+64g64g+64g64

% solo B (4/4)
[3 m14 m15 ] m14 m16
m17 m18 m17 m19 m17 m18 m17

% transition B (7/4)
m23

% four-to-the-floor A (4/4)
[14 m26 ]
m32

% four-to-the-floor B (4/4)
[3 m26 m36 ]
o1 v6 c16>c32r32<c16f32r32a+32r32>c16<f16>d+16e16c32r32<g16f32r32>d+32r32c16e16>c16
[3 m26 m36 ]
r1

% bass solo (4/4)
m37
v6 o1 c16.r32e16r16f16>d+32r32<f+8g32r32g32r32a16>f+16<a+8b16a+64a64g+64g64
f+64f64e64d+64>g64r32.c16<c16r16c16>c64r32.<c16>c64r32.
a+32r32<a16>g32a32<a+32r32>d+16<d+16f32g32
m37
v6 o1 c16r16c16>g32r16.c16g64r32.<c16>a16r16<c16>a+16r16c16>c16<<c16
>a32g32f32f+32g16f+16g16f+16g16a32r32<a16>g32r32a16<a+16>>d+32r32<a+16
<b16>b16
<c16r8a+64>c32.r16<a+64>c32.<a32r32a+16r16a+32r32>c16<a+32r32d+16e16
```

```
a+32r32d+16
r8c16>c16r16c16f+16g16>c32r32d+16e16<a+16c32r32d+32f32r16<a+16
c8>c32r32<e8>e32r32<f16>f16f+32r32<f+16>a16g16<g16>a+16<a+16b16
c16r64>g64g+64a64a+32>c32<a+32g32f16g16c16<a+16>d16d+32.r64e32.r64c16
<a32a+16f+32g32.r64d+16
a+32.r64a+32.r64a32.r16r64a+32.r16r64a32r32a+16>c32.r64<a+32r32>c16<a+32
r32d+16e16a+32r32d+16
a+16c32r32a+16>d+32r32>c32r32<<a+32r16.g16r16a+32r32>c32.r64<f16g+16
a16e16d+16
e32.r64f32.r64f+32.r64g32.r16r64a32.r64a+32.r64>g32r16.<b32.r64>c32.
r16r64<a+16>d16d+16e16
r16<c16>f16r16<c32r32>f+16r16g16<g32r32>g+16<g+32r32>a16<a+16>a+16<b32
r32>b16>

% stutter transition (4/8)
v6 c8. v5 c8. v4 c8. v3 c8. v2 c4
[6 m3 ] v3 [8 o4c64e64<d64g64 ]

% movement II intro (4/4)
v2 [128 o2c64>c64<c64>c64 ]
v4 b32>c2&c8.&c32 [4 c+64c64<b64>c64 ]
d2d+32e8.&e32c4
<f64f+64g2&g8.&g32 [4 g+64g64f+64g64 ]
e32f2&f8.&f32c4
v3 <<c2. v2 c2.
v4 >>c4c+32d8.&d32
v3 <c2. v2 c2.
v4 >c4c+32d8.&d32
e2 [4 f64e64d+64e64 ] g4
>c4<f16g8.>f32g8.&g32>d32e8.&e32
<a64g64f2&f8.&f32 [4 f+64f64e64f64 ]
v2 f2 v6 <<g32g+32g16c+32d32c16<f32.r64g16d16e32r32

% movement II main (4/4)
[2 [18 m40 ] r8 [5 m41 ] ]

% movement II solo (4/4)
m47 m47
v6 o1 e16r4r16e16r4e16r16>c64r32.<e16>c64r32.
<e16r4r16e16r16b16>c16d16<g16e16a+16d+8
d16r4r16d16r8.>c64r16.r64c16<f16d16c16
g8r16>g64r8r32.<g16r16>f16g32r32<g8>f16<f16>d32r32f16
<c8r16>c16r8<c16r8.>c8<b16r16a+4.
g8 v5 g8 v6 a+8 v5 a+8 v6 >c8 v5 c8
m47 m47
v6 o1 e16r8e16r8b16r8.>e8<a4
v5 a4 v6 e8 v5 e8 v6 >g+16 v5 a8. v4 a8 v3 a8
v6 d8 v5 d8r8 v6 d16r4.r16<g8
v5 g4 v4 g4 v5 >>g32 v6 g+16. v5 g+8 v4 g+8 v3 g+8

% transition D
o1 v6 c#1
g+8>g+16r16<a8>a16f+16g+16a+32r32<a+16a16b8.a64f+64d+64c+64
c8r16c+8r16d8r16d+8r16e8r16f8
r16f+2&f+8>a+32r32b16e16<b16
>f16g16<g2.&g8r4

% solo C (4/4)
m14 m15 m14 m16
m17 m18 m17 m19 m17 m18 m17

% main theme reprise (4/4, 7/4)
[2 r1r2.
o5 v6 g32<g32<g32<g32<g32f+64f64e64d+64d64c+64
>c64<c32.r8>c64<c32.r8>c64<c32.r16>>c64<c32.r8c64<c32.r8>c64<c32.r8
>c64<c32.r8>c64<c32.r16>>g+64<g+16.&g+64>c64<c32.r16f64<f16.&f64 ]
m40 m40

% final stretch (4/4)
m53
d1 [8 d16 r16 ]
m53
a1 [8 a16 r16 ]
m53
v6 o1 g8. v5 g2&g32a64>e64g4

% ending stutter
v4 g8 v3 g8 v2 g4
```

```
[6 m3 ] o3
v4 [32 g32>g32< ]
v3 [16 g32>g32< ]
v2 [16 g32>g32< ]
v1 [16 g32>g32< ]
r1r1

%~~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL D %~~~~~~~~~~~~~~~~~~~~~~~~~~%

@ t46 r4

% stutter intro (4/8)
r1r1r1

% Cmin7 arp intro (3/8)
[19 r4. ]
d4.d4.e8e16e16
e16e32e32d4.d8e8d+64d+16.r64d+16d+32d+32
d4.d8r4.e32e32e32e32
d8e16r16e16e64e64e64e64d8d+16d+16c+16d+64d+64d+16.d16d+16d+16

% drum solo Cmin7 (4/4)
m6 m7 m8

% drum solo C#min7 (7/8)
d16e16c16d16d+16e16e16d+64d+32.e16d+16d16e16d+16c16d16e16
c16d16d+16e16e16d+64d+32.e64e64e64e64d+16d16e16d+64d+32.e32e32d16e16c16d16
c+32c+32e16e16d+64d+32.e16d16d+16e16d+16c16d16d16d64d32.d64d32.d+16e16
d16d+64d+32.e16d+16d16d+16d+64d+32.d+16d+16d+64d+32.
m6 m7

% Cmin7 bass groove (4/4), F#13 bass groove (4/4)
[7 m13 ]

% transition A (4/4)
d16e16d16e16d+16r2r16e64e64e64e64d16
e16d+16d16e32e32d+16d16c+16d+16d16e64e64e64e64d+64d+32.d16d+16c16d16d+64d+32.
d+1r2.
d16e64e64e64e64d+16e32e32

% solo A (4/4)
[7 m13 ] m20

% transition B (7/4)
m24

% four-to-the-floor A (4/4)
[14 m25 ]

% transition C
d8e8e8d16e8e16e8d+8e16e8.
d4d16d+8.d+8.d+8.

% solo B (4/4)
[11 m13 ] m20

% transition B (7/4)
m24

% four-to-the-floor A (4/4)
[14 m25 ]
r1

% four-to-the-floor B (4/4)
[14 m25 ]
t53 d+16e32e32c+16d+64d+32.e16d+16c+16d+16d+16d16d+16e32c+32e16d+64d+32.c16d+16

% bass solo (4/4)
[8 m13 ]

% stutter transition (4/8)
t46 r1 r1 r1

% movement II intro (4/4)
t56 [19 r1 ] m38

% movement II main (4/4)
[2 [18 m39 ] r1 m38 ]
```

```
% movement II solo (4/4)
t56 [12 m39 ]
d16r8e32e32d16e16d16r16e16e16d16r8.d+64d+32.r16
r1

% transition D
t46
m38
d8e8d8d16d16d16e16e16d+64d+32.d16e32e32d+64d+32.d+16
[5 d8e16 ] d8
r16d8r4d8r8d8r8
d+16d+16d8r4d8r8d8r8
d16d+32d+32d+64d+32.c16

% solo C (4/4)
[7 m13 ] m20

% main theme reprise (4/4, 7/4)
[2 r1 r1 d8.d16e8d8d+64d+16.&r+64e16d16e8d8
c16d8.d+64d+16.&r+64e8d8d+64d+16.&r+64 ] m13

% final stretch (4/4)
m6 m7 m8 m6
t66 d1

% ending stutter (4/8)
t46
[9 r1 ]

%~~~~~~~~~~~~~~~~~~~~~~~~~~%   MACRO   %~~~~~~~~~~~~~~~~~~~~~~~~~~%

% macro #01 - channel A: stutter Csus (4/8)
@ o4 v5 [8 r64 c32. ]

% macro #02 - channel B: stutter Csus (4/8)
@ o3 [8 v4 f32>f64< v3 f64 ]

% macro #03 - channel C: stutter Csus (2/8)
@ o3 [4 v3 g32> v4 g32< ]

% macro #04 - channel A: arp Cmin11 (3/8)
@ o3 v4 f16c16>c16<< v3 a+16> v4 g16d+16

% macro #05 - channel B: arp Cmin11 (3/8)
@ o4 v6 c32< v3 c32< v4 a+32>> v3 c32< v5 g32< v3 a+32> v5 d+32 v3 g32 v5 f32 v3 d+32 v5 c32 v3
    f32

% macro #06 - channel D: kit solo #1 (4/4)
@ d16e16c16d16d+16e16e16d+64d+32.e16d+16d16e16d+16c16e16d+64d+32.
  d16d+16e16d+16e16e64e64e64e64c+16e16c+16e16c16d+16e16d16d+16e32e32

% macro #07 - channel D: kit solo #2 (4/4)
@ e16d16e32e32c16d+16e16d16d16e16e16d16d16e32e32d+64d+32.e16d+32d+32
  d16e16d16e16d+16e16e16d+16c16d+16d+64d+32.e16e32e32d+64d+32.e16d+64d+64d+64d+64

% macro #08 - channel D: kit solo #3 (4/4)
@ d16e16c16d16c16d+16d16d+64d+32.e16d+16d16e16d+16c16e16d+64d+32.
  [4 d32e32 ] d+16d16d+64d+32.c+64c+64c+64c+64 [4 d32e32 ] d+16 [3 c64c64c+64c+64 ]
  e16d+16e32e32c16d+16e16e32e32d+16c64c64c64c64c+64c+64c+64c+64d16d+64d+64d+64d+64e32e32d+64d+32.
      e16d+32d+32
  [2 d64d64d64d64e64e64e64e64 ] d+64d+32.e64e64e64e64e16d+16c16d+16d+16c16d+64d+32.c+64c+64c+64c
      +64d64d64d64d64d+64d+64d+64d+64

% macro #09 - channel A: arp C#min11 (7/16)
@ o3 v4 c+16e16>c+16<< v3 b16> v4 g+16e16f+16

% macro #10 - channel B: arp C#min11 (7/16)
@ o4 v6 c+32< v3 e32< v4 b32>> v3 c+32< v5 g+32< v3 b32> v5 e32 v3 g+32 v5 f+32 v3 e32 v5 c+32 v3
    f+32 v5 e32 v3 c+32

% macro #11 - channel B: sweep (3/16)
@ o3 v6 b64a+64a64g+64g64f+64f64e64d+64d64c+64c64

% macro #12 - channel C: sweep (3/16)
@ o2 v6 c64<b64a+64a64g+64g64f+64f64e64d+64d64c+64

% macro #13 - channel D: main groove (4/4)
```

```
@ d16e16d16e16d+16e16 [2 e16d+64d+32.e16d+16d16e16d+16e16 ] e16d16e16e16d16e16d+16d16d+16d+16

% macro #14 - channel C: bass groove Cmin7 #1 (4/4)
@ o1 v6 c16r16c16 [3 r8>c16r16<c16r8c16 ]
>c32r32d+32r32<c16a+16>c16

% macro #15 - channel C: bass groove Cmin7 #2 (4/4)
@ o1 v6 c16r16c16 [2 r8>c16r16<c16r8c16 ]
r8c16r16g16a+32r32a+32r32a16r16a+16>d+16<d+8

% macro #16 - channel C: bass groove Cmin7 #3 (4/4)
@ o1 v6 c16r16c16 [2 r8>c16r16<c16r8c16 ]
d+8c16f8c16f+8c16g8a+16>c16

% macro #17 - channel C: bass groove F#13 #1 (4/4)
@ o1 v6 f+16 o3 f+32r32 o1 f+16r16 o3 f+32r32<f+16r16 [2 o1 f+16 o3 f+32r16. ] <e16f+16<f+16

% macro #18 - channel C: bass groove F#13 #2 (4/4)
@ o3 v6 r16f+32r32 o1 f+16r16 o3 f+32r32<f+16r16<f+16 o3 f+32r32<e32r32<f+16>f+32r32a32r32<f+16>
    e16f+16

% macro #19 - channel C: bass groove F#13 #3 (4/4)
@ o3 v6 r16f+32r32 o1 f+16r16 o3 f+32r32<f+16r16<f+16 o3 f+32r32<e32r32d+16r16e32r32d+16e16f+16

% macro #20 - channel D: main groove ending (4/4)
@ d16e16d16e16d+16e16e16d+64d+32.e16d+16d16e16d+16e16e16d+64d+32.

% macro #21 - channel A: transition B (7/4)
@ o5 d32.r64f+32.r64d32.r64c16.&c64r64<a+32.r64>c+32.r64f32.r64c+32.
  r64<b16.&b64r64a32.r8r64>c32.r64e32.r64
  c32.r64<a+16.&a+64r64g+32.r64 o3 [8 v2 g128>g128< ] [8 v4 g128>g128< ]
  [8 v5 g128>g128< ] v7 [6 g128>g128< ] f+128f128e+128d+128

% macro #22 - channel B: transition B (7/4)
@ o3 v6 [2 f+64d64<b64> ] v4 [2 f+64d64<b64> ]
  v6 [2 f64d64<a+64> ] v4 [2 f64d64<a+64> ]
  v6 [2 f64c+64<a+64> ] v4 [2 f64c+64<a+64> ]
  v6 [2 e64c+64<a64> ] v4 e64c+64<a64>e64c+64<a64r8
  v6 [2 >e64c64<a64 ] v4 [2 >e64c64<a64 ]
  v6 [2 >d+64c64<g+64 ] v4 [2 >d+64c64<g+64 ]
  r64 v3 o4 [8 v2 g128>g128< ] [8 v4 g128>g128< ]
  [8 v5 g128>g128< ] v7 [6 g128>g128< ] r64

% macro #23 - channel C: transition B (7/4)
@ o1 v6 [2 g16r16>g32r32< ] f+16r16>f+32r32<f+16r16>>c16<f16f+16<f16r16
  >f32r32<f16r8r64. o4 [8 v3 g128>g128< ] [8 v4 g128>g128< ]
  [8 v5 g128>g128< ] v6 [6 g128>g128< ] r128

% macro #24 - channel D: transition B (7/4)
@ d16e32e32d+16d16e16e16d16e16d+16d16e16d16d+64d+32.d+16d16e16
  d+16d16e16d+16r4. [16 e128 ]

% macro #25 - channel D: four-to-the-floor kit (4/4)
@ d16e16c+16e16d+16e16c+16e16

% macro #26 - channel C: four-to-the-floor bass (4/4)
@ o1 v5 c16> v6 c32r32 v7 <c16f32r32 v5 a+32r32> v6 c16 v7 <f16c16

% macro #27 - channel A + B: arp C7sus4 (4/4)
@ o3 [4 v2 >c64<g64>f64<a+64 v3 >c64<g64 v4 >f64<a+64 v5 >c64<g64 v6 >f64<a+64 r16 ]

% macro #28 - channel A + B: arp C7 (4/4)
@ o3 [4 v2 >c64<g64>e64<a+64 v3 >c64<g64 v4 >e64<a+64 v5 >c64<g64 v6 >e64<a+64 r16 ]

% macro #29 - channel A + B: arp Cm/Fmaj/D#maj (4/4)
@ v6 c64d+64g64 v5 [4 c64d+64g64 ] v3 [3 c64d+64g64 ]
  v6 c64f64a64 v5 [4 c64f64a64 ] v3 [8 c64f64a64 ] r64
  v6 g64a+64d+64 v5 [4 g64a+64d+64 ] v3 [3 g64a+64d+64 ]
  v6 f64a64c64 v5 [4 f64a64c64 ] v3 [8 f64a64c64 ] r64

% macro #30 - channel B: arp F#9/F#maj/F#2sus4 (4/4)
@ v6 c+64e64g+64 v5 [4 c+64e64g+64 ] v3 [3 c+64e64g+64 ]
  v6 c+64f+64a+64 v5 [4 c+64f+64a+64 ] v3 [8 c+64f+64a+64 ] r64
  v6 g+64b64e64 v5 [4 g+64b64e64 ] v3 [3 g+64b64e64 ]
  v6 f+64a+64c+64 v5 [4 f+64a+64c+64 ] v3 [8 f+64a+64c+64 ] r64

% macro #31 - channel B: arp chromatic rise (4/4)
@ o3 v6 c+64e64g+64 v5 [4 c+64e64g+64 ] r64
```

```
  v6 d64f64a64 v5 [4 d64f64a64 ] r64
  v6 d+64f+64a+64 v5 [4 d+64f+64a+64 ] r64
  v6 e64g64b64 v5 [4 e64g64b64 ] r64

% macro #32 - channel C: bass fill transition (4/4)
@ o3 v6 c16<c32r32<c16f32>c32a16a+16>d+32r32c16<f+32f32d+16g16<a+16>c32r32<f16>g64a+32.<b32.r64

% macro #33 - channel A: C7 (F) funk Chord (4/4)
@ o4 [2 v4 f64r32.f64r32. v5 f32r32 ] v4 f64r32.f64r32.f64r32. v5 f32r32 v4 f64r32. v5 f32r32 v4
     f64r32.f64r32. v5 f32r32 v4 f64r32.

% macro #34 - channel A: C7 (E) funk Chord (4/4)
@ o4 [2 v4 e64r32.e64r32. v5 e32r32 ] v4 e64r32.e64r32.e64r32. v5 e32r32 v4 e64r32. v5 e32r32 v4
     e64r32.e64r32. v5 e32r32 v4 e64r32.

% macro #35 - channel B: C7 (A#) funk Chord (4/4)
@ o3 [2 v4 a+64r32.a+64r32. v5 a+32r32 ] v4 a+64r32.a+64r32.a+64r32. v5 a+32r32 v4 a+64r32. v5 a
     +32r32 v4 a+64r32.a+64r32. v5 a+32r32 v4 a+64r32.

% macro #36 - channel C: C7 (C) funk Chord (4/4)
@ o4 [2 v4 c64r32.c64r32. v5 c32r32 ] [4 v4 c64r64 ]

% macro #37 - channel C: bass solo lick (4/4)
@ o1 v6 c16r8.c16r16>c16<c16r16>d+16<c16g16a+16>c16<c32r32>>c32r32
<d+16e16<c16r16>a+64r32.c16a+64r32.<c16>d+32r32a+16<c16>c32r32d+32r32<c16a+16>c16

% macro #38 - channel D: short fill (4/4)
@ e4e4e8d16c16d+16e16e32e32d+16

% macro #39 - channel D: chill groove (4/4)
@ d16e16c+16d16e16e16d16c+16e16e16e16e16d+16e16e16e16

% macro #40 - channel C: movement II bass pedal (4/4)
@ o2 v6 c64<c16.r16r64>c64<c32r8r64>c64<c32r8r64>c64<c32r16r64>>c64<c32r16r64c64<c32r64>b64<b32.>>
     c64<c32.

% macro #41 - channel B + C: movement II arp C2 (4/4)
@ o4 v5 e32 v3 e32 v5 c32 v3 c32 v5 d32 v3 d32 v5 <g32 v3 g32 v5 >e32 v3 e32 v5 c32 v3 c32

% macro #42 - channel A: main melody #1 (4/4)
@ o3 v6 g+64g8.&g32.v4 g8 v3 g8 v6 f64e8.&e32. v4 e8 v3 e8
  v6 >c+64c8.&c32. v4 c8 v3 c8 v6 d+64d8.&d32. v4 d8 v3 d8
  v6 <f32f+64g32.f+64f8&f64 v4 f4 v3 f4 v2 f4

% macro #43 - channel A: main melody #2 (4/4)
@ o3 v6 a+64a8.&a32. v4 a8 v3 a8 v6 g64f+8.&f+32. v4 f+8 v3 f+8
  v6 >f64e8.&e32. v4 e8 v3 e8 v6 d+64d8.&d64e64g64a4

% macro #44 - channel A: main melody #3 (4/4)
@ o3 v6 f32f+64g32.f+64f8&f64 v4 f4 v3 f4 v2 f4
  o1 v3 g4 v4 g4 v5 g4 v6 g4

% macro #45 - channel A: movement II arp C2 (3/16)
@ [3 o5 v6 d64<g64e64 v4 >d64<g64e64 v3 >d64<g64e64 v2 >d64<g64e64 ]

% macro #46 - channel A: movement II arp Fmin6 (3/16)
@ [3 o5 v6 d64<g+64f64 v4 d64<g+64f64 v3 d64<g+64f64 v2 d64<g+64f64 ]

% macro #47 - channel C: bass Fmaj (4/4)
@ o1 v6 f16r4r16f16r4.f16>d64r32.c16

% macro #48 - channel B: chord #1 (4/4)
@ o3 [2 [2 v6 >e64<a64f64 v5 >e64<a64f64 v4 >e64<a64f64 v3 [5 >e64<a64f64 ] ] r4 ]

% macro #49 - channel B: chord #2 (4/4)
@ o3 [2 v6 b64g64e64 v5 b64g64e64 v4 b64g64e64 v3 [5 b64g64e64 ] ]

% macro #50 - channel B: chord #3 (4/4)
@ o3 [2 v6 a64f64d64 v5 a64f64d64 v4 a64f64d64 v3 [5 a64f64d64 ] ]

% macro #51 - channel A: main melody #4 (4/4)
@ o3 v6 a+64b64>c4.&c16. [8 c+64c64<b64>c64 ]

% macro #52 - channel B: arp c#maj9 (3/16)
@ o4 v6d+128<v5d+64.>v6f128<v5f64.>v6g+128<v5g+64.o5 v6d+128<v5d+64.>v6f128<v5f64.>v6g+128<v5g+64.

% macro #53 - channel C: bass C#maj9 (4/4)
@ o1 v6 c#4 v5 c#2. v6 [8 c#16 r16 ]
```

```
% macro #54 - channel A: C#Maj9 Run (4/4)
@ o5 v6 c+16c16<g+16f16d+16c+16c16<g+16f16d+16c+16c16<g+16f16d+16c+16
```

6.2 paganinis-been-at-the-bins.mmml

```
%====================================================================%
% TITLE      : Paganini's Been At The Bins
% COMPOSER   : Blake 'PROTODOME' Troise
% PROGRAMMER : Blake 'PROTODOME' Troise
% DATE       : 3rd February 2019
% NOTES      : Oh come on, not again Paganini. If you keep leaving
%              the bags on the side of the road, he's going to get
%              in there...
%
%              A cool little piece exploring unusual time signatures,
%              classical writing and 1-bit instrumental techniques.
%====================================================================%

%~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL A %~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% paganini-esque introduction
v4 m25 r8 t45

% fade-in
[4 o3v2c32rv1[3cr]v2crv1crv2[3cr][2v1crcrv2cr]cr ]
v3c32rcrv2[9cr]v3crv4crv5crv6crv7cr

% proper introduction
m2 m2

% section A (arp)
[2
  [4 m2 ]
  [3 o3v4d+32v2d+v4c+v2c+v4g+v2g+v4<g+v2g+v4>d+v2d+v4fv2fv4g+v2g+v4<g+v2g+ ]
  o3v4d+32v2d+v4c+v2c+v4g+v2g+v4<g+v2g+ m2
]
m2

% section B1
m14 m14
m15 r64 % solo echo
m14 m14

% section B2
[2
  v7o5c64<b64g64e64c64<b64g64e64c64r32.>e64c64<b64g64r8>e64c64<b64g64
  v5[6o4e64c64<b64g64]r16

  v7o4e64d64<b64g64r8>e64d64<b64g64r8
  >e64d64<b64g64v5[3o4e64d64<b64g64]
  v7o5c32<b32g32e32d32c32<b32a32

  v7>f64d+64c64<g64r8>f64d+64c64<g64r8>f64d+64c64<g64
  v5[6o4f64d+64c64<g64]r16

  v7o4g64d+64c64<a+64r8>g64d+64c64<a+64r8>f64d64c64<g64
  v5[5o4f64d64c64<g64]
  v8[2o4f64d64<b64g64]
]

v1[2o4b64d+64<b64g+64v2] % .'.'.
v3[2o4b64d+64<b64g+64v4] % blblb
v5[2o4b64d+64<b64g+64v6] % bLbLb
v7[2o4b64d+64<b64g+64v8] % BLBLB

% section B3
[2 % wow, this 4 bar segment wasn't worth the memory cost
  v7[2m17r16v5]
  v3[2m17r16v2] m17
  v7[2m18r16v5]
  v3[2m18r16v2] m18
  v7[2m19r16v5]
  v3[2m19r16v2] m19
  v7[2m20r16v5]
  v3m20
  [2v7o4[2f64e64c64g64]r16v6]
]
```

```
% ending
o3g32g+32f16o5c16<g+16<b16g16>d64>d32.b64<b32.>c32r32<g32r32d+16<g16
v8[2m17r16v7]v6[2m17r16v5]v4[2m17r16v3]v2[2m17r16v1]o1c16c2

% shh
r4

%~~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL B %~~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% paganini-esque introduction
r16 v3 m25 r16

% fade-in
r1r
[2 v2o4d32[2v1o3fv2gv1>dv2cv1<gv2fv1>cv2d]v1<fv2gv1>dv2gv1<gv2>fv1gv2ev1fv2cv1ev2<gv1>cv2<fv1g ]
v2o4d32v1<fv2gv1>dv3cv2<gv3fv2>cv4dv2<fv4gv2>dv4cv2<gv4fv2>cv4dv2<fv4gv2>dv4gv2<gv4>fv2gv4ev2fv4c
     v2ev5<gv3>cv5<fv3g

% proper introduction
m3

% section A (bass)
[2
  [3 m3 ]
  [4 o1v5a+16v3a+16 [2 v4o4c16v2c16v4<d16v2d16v5<<a+32v3a+32 ] v4o4c16<c16 ]
  m3
  [2 o2v5c+16v3c+16 [2 v4o4c+16v2c+16v4<d+16v2d+16v5<c+32v3c+32 ] v4>>c+16<c+16 ]
]

[2 m3 ]

% section B1
m13 m13
r64 m15 % solo
m13 m13

% section B2
[2 m5 v6 [2 o3 f64g>dfg>dfg32f64d<gfd<gf ] ]
r2

% section B3
[2 % another massive waste of space
  v8[2m21r16v6]
  v4[2m21r16v3]
  v1m21
  v8[2m22r16v6]
  v4[2m22r16v3]
  v1m22
  v8[2m23r16v6]
  v4[2m23r16v3]
  v1m23
  v8[2m24r16v6]
  v4m24
  v8[2o3e64f64a64>c64e64g64>c64f64r16v7]
]

% ending
r2.v8[2m21r16v7]v6[2m21r16v5]v4[2m21r16v3]v2[2m21r16v1]r16r2

% shh
r4

%~~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL C %~~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% paganini-esque introduction
r8 v2 m25

% fade-in
r1rr
[3 o3v2d32v1dv2cv1cv2gv1gv2<gv1gv2>dv1dv2ev1ev2gv1gv2<gv1g ]

% proper introduction
[15 m1 ]
```

```
% section A1 (melody)
[2 m5 m1 ] m6 m5 m1 m4
m1 m1 o3v4d32v2dv4cv2cv4gv2gv4<gv2g

v7 [2 m7 m8 v6 ] v5 m7 m8 v4 m7 % arp

% section A2 (melody)
[2 m5 m1 ] m6 m5 m1
o3v6c8.v4c8.v2c8v6f8v2f16v6g+8v2g+16v6a+8.
v4a+8.v2a+8v6g+8.a64a+b>cc+dd+eff+gg+

v6[18 o3c64e64g64>f+64d64<b64  ] c64e64g64>f+64
  [18 o4d64c64<d+64f64g+64>g64 ] d64c64<d+64f64
  [18 o3a64f+64d64>a64f+64d64  ] f+64d64>a64
v7[10 o4g64d64<a+64g64d64>a+64 ] g64d64<a+64g64d64
v8[7  o4b64g64d+64<b64g64d64   ] b64g64d+64<b64g32

% section B1
[4 m10 ]
[2
  o1c8>c64r32.<c16r8>c8<v8[6c32r32]v7g8
  r8>c64r16.r64g8<v8[6g32r32]v7c+8>c64r32.<c+16
  r8>c+8<v8[6c+32r32]v7g+8r16>c64r16.r64c64r32.
  g+16.g64d64<g16a32r32f32r32g16>c16g16
]

% section B2
m10 m10
r2

% section B3
[2 v7o2c32<c16.r2>c32<c16.r8>g32<g16.r2.>c+32<c+16.r2>c+32<c+16.r8>g+32<g+16.r4.>g32<g16.r4 ]

% ending
r2.r16v8[2m21r16v7]v6[2m21r16v5]v4[2m21r16v3]v2[2m21r16v1]r2

% shh
r4

%~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL D %~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% paganini-esque introduction & fade-in & proper introduction & section A1
[58 r1 ]
d+64d+32.r16d16d+16e16d+64d+32.d16r16d+16d+16

% section A2
[4 m9 ]
d16rerdrerd+rererdreed+64d+32.r16dd+ed+64d+32.d16rd+d+
m9 m9

% section B1
[8 m11 m12 ]
[2[3 d16e16e16d16c+16e16d+64d+32.r16[6 c+64e32. ]] m12 ]

% section B2
[2
  m16
  m11 m12
]
d+16c16e32e32c+64c+64c+64c+64d+64d+32.e16c+16d+64d+32.

% section B3
[4 m16 ]

% ending
r2.r1r1r16

% shh
r4

%~~~~~~~~~~~~~~~~~~~~~~~~~%   MACRO   %~~~~~~~~~~~~~~~~~~~~~~~~~%

% m1: C arp #1
@ o3v4d32v2dv4cv2cv4gv2gv4<gv2gv4>dv2dv4ev2ev4gv2gv4<gv2g

% m2: A arp #1
```

```
@ [7 o3v4c32v2cv4gv2gv4<gv2gv4>dv2dv4ev2ev4gv2gv4<gv2gv4>dv2d ]

% m3: B bass #1
@ [3 o2v5c16v3c16 [2 v4o4c16v2c16v4<d16v2d16v5<c32v3c32 ] v4>>c16<c16 ]
  o2v5c16v3c16v5>>c16v3c16v5<d16v3d16v5<c32v3c32v5>>c16v3c16v5<<g16v3b64v5>c32.<c16>d32e32<<b32v3
       b32

% m4: C tune #3
@ o4v6c64<c8&c32.v4c8.v2c8v6>f64<f16.&f64v2f16v6>g+64<g+16.&g+64v2g+16v6>a+64<a+8&
  a+32.v4a+8.v2a+8v6>g+64<g+8&g+32.v4g+8.v6>g64<g8.&g32.v4g4

% m5: C (& B) tune #1
@ o4v6c64<c8&c32.v4c8.v2c8v6>e64<e16.&e64v2e16v6>g64<g16.&g64v2g16v6>b64<b8&
  b32.v4b8.v2b8v6>c32>d64<d8&d64v4d8.v6>c64<c8.&c32.
  v5c4v4c4v3c2

% m6: C tune #2
@ o4v6c64<c8&c32.v4c8.v2c8v6>f64<f16.&f64v2f16v6>g64<g16.&g64v2g16v6>>c64<c8&
  c32.v4c8.v2c8v6>d64<d8&d32.v2d8.v6e32>f64<f8.&f64
  v2f8v6d32>e64<e8.&e64v2e8v6>c64<c8.&c32.v4c4v3c4v2c4

% m7: C arp #2a
@ [2o5g64edc<gedc]

% m8: C arp #2b
@ [2o4g64edc<gedc]

% m9: D kit #1
@ d16rec+d+rererc+ed+rerdrec+d+rerec+red+eed
  r8e16c+d+rereec+ed+redd+d+64d+32.e16c+d+ed+d+

% m10: C bass #1
@ v7o1c8>c64r32.<c16r8>c8r16c64r32.<c8>c16.<b64a64g8
  r8>c64r16.r64g8r16c64r32.<g8>f32g16.<c+8>c64r32.<c+16
  r8>c+8r16c64r32.<c+16>g+16<c+16d64e64f64g64g+8r16>c64r16.r64c64r32.
  g+16.g64d64<g16a32r32f32r32g16>c16g16

% m11: D kit #2
@ d16eedc+ed+64d+32.e16eedede

% m12: D kit #3
@ d16eedeed+64d+32.e16eeded+64d+32.d16

% m13: B arp #1
@ [2 v7o3d+64g64>c64d+64g64>c64d+64g32d+64c64<g64d+64c64<g64d+64 ] r4.
  [2 o2b64>d64g64b64>d64g64b64>d32<b64g64d64<b64g64d64<b64] r4.
  [2 o3f64g+64>c+64f64g+64>c+64f64g+32f64c+64<g+64f64c+64<g+64f64 ] r4.
  [2 o3c64d+64g+64>c64d+64g+64>c64d+32c64<g+64d+64c64<g+64d+64c64 ]
  [2 o5f64e64c64<a64g64f64e64c64<a64g64f64e64 ]

% m14: A arp #2
@ v8o4c64d+64<g64a+64  v6[7o4c64d+64<g64a+64]
  v8o5g64a+64r32       v6[5g64a+64r32]
  v8o3g64b64>d64f64    v6[7o3g64b64>d64f64]
  v8o5g64b64r32        v6[5g64b64r32]
  v8o4c+64f64<g+64>c64 v6[7o4c+64f64<g+64>c64]
  v8o5f64g+64r32       v6[5f64g+64r32]
  v8o3g+64>c64d+64g64  v6[7o3g+64>c64d+64g64]
  v8f64e64c64g64       v6[5f64e64c64g64]

% m15: A (& B) solo
@ v7o4a+64>c32.<g16d+16<g16d+16f16g16>c32r32d+16c16<g16v8d+32r32c16<g16
  v7b16>g16
  d16b32.>c64c+64d32.g16b16>d16g16b16g16d16<b32r32f64g32.f16c+16<g+16f16
  >>c64f32.c+16<g+16f32.r64c+16<g+32r32f16g+32r32c+32.r64f16g+32r32>c16
  d64d+32.g+16>c16d+16
  g16f16c16<g16d+64f32.c16<g16f16v8c32d+32g32>c32d+32g32>c32d+32g32d+32
  c32<g32d+32c32<g32d+32
  v7c32r32d32r32d+32r32f32r32g32r32b32r32v8<g32b32>d32g32b32>d32g32b32
  >d32<b32g32d32<b32g32d32<b32v7>g64<g32r64>a64<a32r64
  >b64<b32r64>>c64<c32r64>d64<d32r64>d+64<d+32r64v8c+32f32g+32>c+32f32
  g+32>c+32f32g+32f32c+32<g+32f32c+32<g+32f32v7c+16d+16f32r32a+16
  g+16>c+16f64g32.d+16c32r32<g+16g16d+16c16g16a32>c32d32d+32v8>e64<e32
  r64>f64<f32r64>g64<g32r64>c32.

% m16: D kit #4
@ d8.c16e8d8d+64d+16.r64e16d16e8d8
  c16d16e8d+64d+16.r64e8d16e16d+64d+16.r64
```

```
% m17: A arp #3a
@ [2o4c64d+64<g64a+64]

% m18: A arp #3b
@ [2o3g64b64>d64f64]

% m19: A arp #3c
@ [2o4c+64f64<g+64>c64]

% m20: A arp #3d
@ [2o3g+64>c64d+64g64]

% m21: B arp #1a
@ o3d+64g64>c64d+64g64>c64d+64g64

% m22: B arp #1b
@ o2b64>d64g64b64>d64g64b64>d64

% m23: B arp #1c
@ o3f64g+64>c+64f64g+64>c+64f64g+64

% m24: B arp #1d
@ o3c64d+64g+64>c64d+64g+64>c64d+64

% m25: paganini-esque solo (intro taken from the movie crossroads)
@ o2t90c16t70d+g>t60cd+t56g>cd+gd+t55c<gd+c<d+c
  t80dft70gb>t60dt50fgb>d<bt55gfd<bt60gt70f
  t70cet65ga+>t56cega+>c<a+gec<a+ge
  t60cft55g+>cft53g+>cfg+t55fc<g+fc<g+f
  dfg+b>dfg+b>d<bg+fd<bg+f
  t50cgd+>c<t55g>d+cgd+>t80c<g>d+t90cgd+t100b
  t200>c4t40r1
  % let's goooooo!
  t60c16t50<g16t48d+16<g16d+16f16t44g16>c+16d16<b16g16d+16d16t40<b16>d16f16
  d+16g16a+16>d+16g16a+16>d+16g16a+16f16d16<a+16f16d16<a+16f16
  >>e16c16<g16e16c16<g16e16g16c16f16g+16>c16f16g+16>c16f16
  a+16f16d16<a+16f16d16<a+16f16<a+16>d+16g16a+16>d+16g16a+16>d+16
  f16d16<b16g+16f16d16<b16g+16>d+16c16<g16d+16c16<g16d+16g16
  >g+16f16>>c16<g+16<b16g16>>d16<b16>c16<g16d+16<g16c16r8.
  >c32d+32g32>c32d+32r16.<<g32b32>d32g32b32>d32r16<<c32d+32g32>c32d+32g32
  >c32<g32d+32c32<g32d+32c32r16.
  >>c16<g16d+16c16g16d+16c16<f+16g16<b16>d16f+16g16b16>d16f16
  c32d+32g32>c32d+32r16.<<g32b32>d32g32b32>d32r16<<c32d+32g32>c32d+32g32
  >c32<g32d+32c32<g32d+32c32r16.
  c16e16g16>c16<d16f16a+16>f16<g16a+16>d+16g16<b16>d16g16b16
  <c32e32g32>c32e32g32>c32e32c32<g32e32c32<g32e32c32e32f32g+32>c32f32g+32
  >c32f32g+32g32g+32f32c32<g+32f32c32<g+32
  <a+32>d32f32a+32>d32f32a+32>d32<a+32f32d32<a+32f32d32<a+32>d32d+32g32
  a+32>d+32g32a+32>d+32g32a+32g32d+32<a+32g32d+32<a+32g32
  <g+32>c32d+32g+32>c32d+32g+32>c32<g+32d+32c32<g+32d+32c32<g+32>c32d32
  f+32a32>d32f+32a32>d32f+32a32f+32d32<a32f+32d32<a32f+32
  g+16f16>>c16<g+16<b16g16>>d16<b16>c16<g16d+16<g16c16r8.
  >>d+16c16<g16d+16c16g16d+16c16>d16<b16g16d16b16g16d16<b16
  >>d+16c16<g16d+16c16g16d+16c16>d16<b16g16d+16d16d+32d32d+32d32d+32d32
  >g16e16c16<g16e16c16<a+16g16>>f16c16<g+16f16c16<g+16f16e16
  >>f16d16<a+16g+16f16d16<a+16g+16>>d+16<a+16g16d+16<a+16>d+16<g16a+16
  >>d16c16<b16g+16g16f16d+16d16c16<a+16g+16g16f16f+16g16c16
  g+16f16>>t50c16<g+16<b16t60g16>>d16<b16>t70c16<g16t80e16<t90g16t200c4

%======================================================================%
```

6.3 goose-communications.mmml

```
%====================================================================%
% TITLE      : Goose Communications
% COMPOSER   : Blake 'PROTODOME' Troise
% PROGRAMMER : Blake 'PROTODOME' Troise
% DATE       : 1st September 2018
% NOTES      : .... . .-.. .-.. ---
%              ..-. . .-.. .-.. --- .--
%              --. . . ... .
%====================================================================%

%~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL A %~~~~~~~~~~~~~~~~~~~~~~~~~%

@ o4 r4

% introduction (3/4)
[14 r2. ] r2

% theme A1 (3/4)
m2 m3 m2 m4

% theme A2 (3/4)
m5 r1r4

% theme A1 (3/4)
m2 m3 m2 m4

% theme A2 (3/4)
m5 r4

% theme B1 (4/4)
m15 o3 v6 c+8r16 v3 c+16 v6 e8r16 v3 e16 v6 f+8r16 v3 f+16 v6 e8r16
v3 e16 v6 e8. v5 e8. v6 f+16e16d+8. v5 d+16
m15 m18

v7 [2 o2 a64>c+64e64a64>c+64e64a64>c+32<a64e64c+64<a64e64c+64<a64 ]
[2 o5c+64<b64a64f+64e64c+64<b64a64f+64e64c+64<b64 ]
>a64b64>c+64e64f+64g+64b64>c+64

% theme B2 (4/4)
m21
[10 o3 v5 d64g+64 v7 >c+64 ]
[11 v5 <d64g+64 v7 >e64 ] r64<
[5 v5 c+64a64 v7 b64 ]
[5 o3 v5 c+64a64 v7 >e64 ] <
[11 v5 e64c+64 v7 a64 ] r64
[10 v5 c64e64 v7 a64 ]
[6 v5 o3 c64e64 v7 >e64 ]
[5 o3 v5 c64e64 v7 >f+64 ] r64
[5 o3 v5 g+64<b64 v7 >>g+64 ]
[5 o3 v5 g+64<b64 v7 >>e64 ] <
[6 v5 g+64<b64 v7 >b64 ]
[5 v5 e64<b64 v7 >g+64 ] r64
[21 v5c+64e64 v7 g+64 ] r64
[10 v5 d+64<b64 v7 >f+64 ]
[11 v5 d+64<b64 v7 >g64 ] v5 r64
m21
[10 v5 o3 d64g+64 v7 >e64 ]
[11 v5 <d64g+64 v7 >f+64 ] r64
[5 v5 <c+64a64 v7 >f+64 ]
[5 v5 <c+64a64 v7 >g+64 ]
[11 v5 o3 g+64c+64 v7 >e64 ] r64
[11 v5 o3 c64g+64 v7 >e64 ]
[10 v5<c64a64 v7 >e64 ] r64
[5 v5 <g+64<b64 v7 >>f+64 ]
[5 v5 <g+64<b64 v7 >>e64 ] <
[6 v5 g+64<b64 v7 >b64 ]
[5 v5 e64<b64 v7 >g+64 ] r64
m18

% theme C (4/4)
v6 >f+4g32g+32 v5 g+16r8 v6 e16 v5 e16 v6 <b16 v5 b16 v4 b16 v3 b16
v2 b16 v1 b16 v6 >g+32e32c+32<b32a32r16. v4 >g+32e32c+32<b32a32r16.
v2 >g+32e32c+32<b32a32r16. v6 b16r8b16>f+4f32g32 v5 g16r8 v6 f+16 v5
f+16 v6 e16 v5 e16 v4 e16 v3 e16 v2 e16 v1 e16 v6 g32e32c32<b32a32
r16. v4 >g32e32c32<b32a32r16. v2 >g32e32c32<b32a32r4r16. v6 >a+4b32>
```

```
c+32 v5 c+16r8 v6 <f+16 v5 f+16 v6 c+16 v5 c+16 v4 c+16 v3 c+16r8 v6
f+32d+32c+32<b32a+32r16. v4 >f+32d+32c+32<b32a+32r16. v2 >f+32d+32
c+32<b32a+32r4r16. v6 >g4g+32a32 v5 a16r8 v6 f16 v5 f16 v6 c16 v5
c16 v4 c16 v3 c16 v2 c16 v1 c16 v6 f32d32c32<a+32a32r16. v4 >f32d32
c32<a+32a32r16. v2 >f32d32c32<a+32a32r16. v6 b16r8b16>f+4g32g+32 v5
g+16r8 v6 e16 v5 e16 v6 <b16 v5 b16 v6 >c+16 v5 c+16 v6 e16 v5 e16 v6
a16r16g+32a32g+16f+8a8f+32g+16.r16 v3 g+16 v6 d+32e16.r16 v3 e16 v6 <
b4 v5 b4 v4 b4 v3 b4 v6 >>b4 v5 b4 v4 b4 v3 b4

% ending melody (3/4)
[5 r1 ] r2.
o2 v6 b16r8b16>
[2 e4 v4 e4 v3 e4 v2 e4. v4 e64 v6 e32. v4 d+64 v6 d+32. v4 e64 v6
e32. v4 <b64 v6 b32. v4 >e64 v6 e32. v4 g+64 v6 g+32. v3 f+32 v4 f+32
v5 f+32 v6 f+8&f+32 v4 f+4 v6 g+8 v5 g+8 v6 e4 v4 e4 v6 d+8 v5 d+8 ]

% C#maj13, Amaj13 (4/4)
v4 o5 [128 e64<a64f64> ] [86 d64<a64e64> ] r64

% Gmaj13#11/A, Amaj13#11, A#maj13#11/A
v5 o5 [128 f64c+64<a+64> ]
[64 g64c+64<a+64> ]
[64 g+64d64<a64> ]

% Dmaj9, Fmaj9 (6/8)
[3 [16 c+32<g+32f32> ]
[16 e32<b32g+32> ] ]

[16 c+32<g+32f32> ]
[8 e32<b32g+32> ]
r2 r8. r64

% theme E
m29 m30 m30
m29 m30
v4 o4 [8 c+32g+32d+32 ]

m31
v6 f+16 v8 g+32 v7 g+8&g+32 v5 g+8
f+16 v7 g+32 v6 g+8&g+32 v4 g+8
f+16 v6 g+32 v5 g+8&g+32 v3 g+8
f+16 v5 g+32 v4 g+8&g+32 v2 g+8
r4

m31
o4 v7 f+32g8.&g32 v5 g8 v7 a64b16&b64r32 v7 g16a16g16f+16g16d16 v6 g16b16
v8 a32 v7 a8.&a32 v5 a4 v7 <c+64e64g+64b64>c+64e64f+64g64g+64a32.g+16a16
e16b64>c+32.f+16

% true ending
m32 r1 r2.
r16 o5 v1 e32 v2 e32 v3 e32 v4 e32 v5 e32 v6 e32 m32
r4

r4

%~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL B %~~~~~~~~~~~~~~~~~~~~~~~~~%

@ o3 r4

% introduction & theme A (3/4)
[40 m1 ] m37

% theme A1 (3/4)
[2 m9
o4 v5 [3e64<f+64a64>]
v5 >a32. v4 e64 v3 e32. v6 b64 v5 b32. v4 a64 v3 a32.<
v5 [3e64<f+64a64>]
v5 a32. v4 >e64 v3 e32. v6 <b64 v5 b32. v4 a64 v3 a32. v6 >e64 v5
e32. v4 <b64 v3 b32.
v5 [3e64<f+64a64>]
v5 >b32. v4 a64 v3 a32.<
v5 [3e64<f+64a64>] v5 a32. v4 >e64 v3 e32. v6 <b64 v5 b32. v4 a64
v3 a32.

v5 [3 o3 e64a64>c+64]
o5 v5 a32. v4 e64 v3 e32. v6 b64 v5 b32. v4 a64 v3 a32.
v5 [3 o3 e64a64>c+64]
```

```
v5 a32. v4 >e64 v3 e32. v6 <b64 v5 b32. v4 a64 v3 a32. v6 >e64 v5
e32. v4 <b64 v3 b32.
v5 [3e64<a64c+64>]
v5 >b32. v4 a64 v3 a32.<
v5 [3e64<a64c+64>] v5 a32. v4 >e64 v3 e32. v6 <b64 v5 b32. v4 a64
v3 a32.
m9 ]

% theme A2 (3/4)
r8 [2 o3 v6 a64>c+64<a64>c+64r16 v3 <a64>c+64<a64>c+64r8. ]
[2 v6 <a64>c64<a64>c64r16 v3 <a64>c64<a64>c64r8. ]
[2 o3 v6 g+64b64g+64b64r16 v3 g+64b64g+64b64r8. ]
[2 o3 v6 a+64>c+64<a+64>c+64r16 v3 <a+64>c+64<a+64>c+64r8. ]
[3 v6 <b64>e64<b64>e64r16 v3 <b64>e64<b64>e64r8. ]
v6 <b64>e64<b64>e64r16 v3 <b64>e64<b64>e64r16
m12 r4

% theme B1 (4/4)
[18 m1 ] m19
o2 v5 a8r16 v2 a16 v5 b8r16 v2 b16 v5 >c8r16 v2 c16 v5 c+8r16 v2 c+16
m12 r4

% theme B2 (4/4)
[18 m1 ]
o3 v7 b64a64g64f64d+64c+64<b64a64g+8a16r16b8
v5 o2 a+8r16 v2 a+16 v5 >c+8r16 v2 c+16 v5 <a8r16 v2 a16 v5 b8r16 v2
b16 v6 g+32a+32g+8. v5 g+4 v4 g+4 v3 g+4

% theme C (4/4)
v6 m22 o3
[2 [3 <b64>g64e64 ] r8.r32. [3 <b64>g64e64 ] r4.r16.r64 ]
[2 [3 f+64d+64a+64 ] r8.r32. [3 f+64d+64a+64 ] r4.r16.r64 ]
[2 [3 a64f64d64 ] r8.r32. [3 a64f64d64 ] r4.r16.r64 ]
m22

v6 o5 a16r16g+32a32g+16f+8a8f+32g+16.r16 v3 g+16 v6 d+32e16.r16 v3
e16
o4 a64 v5 a32. v4 >e64 v3 e32. v6 <b64 v5 b32. v4 a64 v3 a32.

% ending melody (3/4)
[17 m1 ]

% C#maj13, Amaj13 (4/4)
[16 v5 o4 e64>e64 v3 <<d64>d64 v5 <a64>a64 v3 e64>e64 v5 <c64>c64 v3
<<a64>a64 v5 <f64>f64 v3 c64>c64 v5 <<g64>g64 v3 <f64>f64 v5 <d64>d64
v3 <g64>g64 ]

[10 v5 o4 e64>e64 v4 <<d64>d64 v5 <b64>b64 v4 e64>e64 v5 <c+64>c+64
v4 <<b64>b64 v5 <f+64>f+64 v4 c+64>c+64 v5 <<a64>a64 v4 <f+64>f+64 v5
< d64>d64v4<a64>a64 ]
v5 o4 e64>e64 v4 <<d64>d64 v5 <b64>b64 v4 e64>e64 v5 <c+64>c+64 v4
<<b64>b64 v5 <f+64>f+64 v4 c+64>c+64 v5 <<a64>a64 v4 <f+64>f+64

% Gmaj13#11/A, Amaj13#11, A#maj13#11/A
[16 v6 o4 f32 v4 <f+32 v6 a+32 v4 >f32 v6 c+32 v4 <a+32 v6 g+32 v4 >
c+32 v6 <d+32 v4 g+32 v6 f+32 v4 d+32 ]
[8 v6 o4 g32 v4 <f32 v6 a+32 v4 >g32 v6 c32 v4 <a+32 v6 g+32 v4 >c32
v6 <d+32 v4 g+32 v6 f32 v4 d+32 ]
[8 v6 o4 g+32 v4 <b32 v6 f+32 v4 >g+32 v6 e32 v4 <f+32 v6 >c+32 v4
e32 v6 <a32 v4 >c+32 v6 <b32 v4 a32 ]

% Dmaj9, Fmaj9 (6/8)
[3 m23 m24 m24 ]
m23 m24
r2 r8.

% theme E
[2 [3 o5 v6 c+32 v4 <<a+32 v6 >f32 v4 >c+32 v6 <g+32 v4 f32 v6 <f+32 v4 >g+32 v6 <a+32 v4 f+32 ]
v6 >c+32g+32
m34
[3 o5 v6 d32 v4 <<b32 v6 >f+32 v4 >d32 v6 <a32 v4 f+32 v6 <g32 v4 >a32 v6 <b32 v4 g32 ]
v6 >d32a32
[3 o5 v6 e32 v4 <c+32 v6 g+32 v4 >e32 v6 <b32 v4 g+32 v6 <a32 v4 >b32 v6 c+32 v4 <a32 ]
v6 >e32b32

[2 [3 v6 o5 g+32 v4 <a+32 v6 >c+32 v4 g+32 v6 d+32 v4 c+32 v6 <c32 v4 >d+32 v6 <a+32 v4 c32 ]
v6 d+32>g32 ] ]
```

```
m33 m34 m35 m33 m33
m33 m34 m35

[3 v6 o5 g32 v4 <f+32 v6 b32 v4 >g32 v6 d32 v4 <b32 v6 d32 v4 >d32 v6 <f+32 v4 d32 ] v6 g32>d32
v6 o3 [2 c+64e64g+64b64>c+64e64a64>c+32<a64e64c+64<b64g+64e64c+64 ]
[2 o5 f+64<a64e64c+64<b64g+64e64c+64 v5 ] o4 v6 f+16c+16g+64a32.>c+16

% true ending
v5 <g+2 v4 g+4. v5>c64<<c64>c+64>c+64<<d64>d64>d+64<<d+64
v6 m36
[5 o4 g+64<g+64<b64>>e64<e64b64 ]
g+64<g+64
[5 o3 d64>g64<g64>d64b64<b64 ]
d64>g64
m36
[5 o5 c64<c64<d+64>g+64<g+64>d+64 ]
c64<c64
[5 o3 c+64>f+64<f+64>c+64a+64<a+64 ]
c+64>f+64
[5 o3 a64>e64>c+64<c+64<e64>a64 ]
a64>e64
[5 o5 d+64<d+64<f+64>>c64<c64f+64 ]
d+64<d+64
[3 m1 ] m37
r4

r4

%~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL C %~~~~~~~~~~~~~~~~~~~~~~~~~%

@ o1 r4

% introduction & theme A (3/4)
[4 r2. ]
[2 o3 v4 e4r8 v3 e4r8 v2 e4r8 v1 e4r8 ]
[31 v4 e16 v3 >e16 v4 <a16 v3 e16 v4 >e16 v3 <a16 v4 e16 v3 >e16 v4
<a16 v3 e16 v4 >e16 v3 <a16 ]
r1r2

% theme A1 (3/4)
[2 [5 m6 ] m7 m6 m6 ]

% theme A2 (3/4)
m10 m10 m11 m11 m10 m10
o1 v7 b4 v6 b4 v5 b4 v4 b4 v3 b4 v2 b4r2

% theme B1 (4/4)
m13 m16
o2 v7 e16 v6 e16r16 v8 >>e64r32. v6 <b16r16 v7 <e16 v6 >b16>e4 v4 e4
v7 <<d+16 v6 d+16>>e16d+16e16<b16 v7 <d+16 v6 d+16>>e16f+8. v4 f+4
v7 <<d16 v6 d16r16 v8 >>e64r32. v6 <b16r16 v7 <d16 v6 d16>>g+4 v4 g+4
v7 <<c+16 v6 c+16r16>>g16g+16e16 v7 <<c+16 v6 c+16>>a4 v4 a4
v7 <<c16 v6 c16r16 v8 >>e64r32.<e64r16.r64 v7 <c16 v6 c16 v5 o5 c16
r16 v6 <b32>c32<b16 v5 a8>c8<a32b16.>e16r16<g+8
o1 v7 b16 v6 b16 o4 e8 v5 d+64d64c+64c64 v4 <b64a+64a64
g+64 v8 e64r8.r32.

v6 <f+8r16 v3 f+16 v6 g+8r16 v3 g+16 v6 a8r16 v3 a16 v6 a+8r16 v3
a+16 v6 b64>c64<b32b2.&b16a+64a64g+64g64d+64d64c+64c64
<b2.&b8.>c64c+64d64d+64

% theme B2 (4/4)
m13 m16 m13
v7 o1 f+16 v6 f+16r4. v7 b16 v6 b16r4.
v7 e16 v6 e16r2r8 v7 >>b16r8b16

% theme C (4/4)
v7 o1 a8r16 v8 o4 e64r32.<e64r16.r64 v7 <<a8r8. v8 o4 e64r32.<e64
r32. v7 <<a32r32>b16a32r32<a8r16 v8 o4 e64r32.<e64r16.r64 v7 <<a8r8
a32r32 v8 o4 e64r32.<e64 v7 <g+32.a16<a16r16>c8r16 v8 >>e64r32.<e64
r16.r64 v7 <c8r8. v8 >>e64r32.<e64r32. v7 <g16c32r32<a16>c8r16 v8
>>e64r32.<e64r16.r64 v7 <c8 v8 e64 v7 e32r64 v8 f64 v7 f32r64 v8 g64
v7 g32. v8 a64 v7 a32r64 v8 >c64 v7 c32. v8 <a+64 v7 a+32r64 v8 b64
v7 b32. v8 g64 v7 g32r64c+8r16 v8 >>d+64r32.<e64r16.r64 v7 <c+8r8
c+32r32 v8 >>d+64r32.<e64r32. v7 <c+32r32>c+16<b32r32c+8r16 v8 >>
d+64r32.<e64r16.r64 v7 <c+8r8. v8 >>d+64r32.<e64 v7 <<b32.>a+16c+32
r32d+32r32c8r16 v8 >>d64r32.<e64r16.r64 v7 <c8r8b32r32 v8 >>d64r32.
<e64 v7 <<b32.>c16>c32r32e32r32<c8r16 v8 >>d64r32.<e64r16.r64 v7
```

```
<c8 v8 >>c+64c64r32<a64g+64r32a64g+64r32>d64r32.<e64r16.r64a64g+64
g64f+64f64e64r32 v7 <<b8r16 v8 >>b64r32.e64r16.r64 v7 <<b8r8b32r32
v8 >>b64r32.e64 v7 <g+32.r16g+16a16<b8r16 v8 >>b64r32.e64r16.r64 v7
<<b8r8>e16 v8 >b64r32.e64 v7 <f+32.r16a32b16.<b8 v6 b8 v5 b8 v4 b8

>>[11 e64c+64a64]
r64 v4 e64r64 v7 a64r64 v4 a64r64 v7 b64r64 v4 b64r64 v6 e64r64 v4
e64r64 v6 a64r64 v4 a64r64 v6 b64r64 v4 b64r64 v5 e64r64 v3 e64r64 v5
a64r64 v3 a64r64 v5 b64 r64 v3 b64r64 v4 e64r64 v2 e64r64 v4 a64r64
v2 a64r64 v4 b64r64 v2 b64r64 v3 <e16a16b16>d+16

% ending melody (3/4)
v4 [4 o3 [32 e64g+64b64 ]
[16 e64a+64>c+64< ]
[16 e64a64>c64< ] ]

% C#maj13, Amaj13 (4/4)
o3 v6 d64d+64e4&e32 v5 e4&e16 v4 e4&e16 v3 e4. v2 e4. r4 r16
[2 v8 o1 g64 v7 g4&g16.&g64r16>g16r8 v3 g16r8 v3 g16 v8 <f64 v7
f16.&f64 v8 g64 v7 g16.&g64r16>g16r8 v3 g16 v8 <g64 v7 g16.&g64 v3 >
g16r8 v7 g16r16 v8 <f64 v7 f16.&f64 ]

[2 v8 o1 e64 v7 e4&e16.&e64r16>e16r8 v4 e16 v8 >>e64r32.<e64r32. v3
<e16 v8 <f64 v7 f16.&f64 v8 e64 v7 e16.&e64r16>e16r8 v4 e16 v8 <e64
v7 e16.&e64 v3 >e16r16 v8 >>e64r32.<e64 v7 <e32.r16 v8 <f64 v7 f16.&
f64 ]

% Gmaj13#11/A, Amaj13#11, A#maj13#11/A & Dmaj9, Fmaj9 (6/8)
[8 v8 o1 c+32 v7 c+4.&c+16.r8 v7 >c+8 v8 >e64r16.r64 v7 <g+8>c+8<g+8
c+8g+8 v8 <c+32 v7 c+4.&c+16.r8 v7 >c+8 v8 >e64r4r16.r64e64r8.r32.e64
r16r32. ]

% theme E
m27 m28
o1 [8 v7 c+32.r64 v6 c+32.r64 ]
[7 v7 d+32.r64 v6 d+32.r64 ] v7 d+32.r64 v6 d+64f64g+64a+64
[8 v7 b32.r64 v6 b32.r64 ]
[8 v7 a32.r64 v6 a32.r64 ]
m28

m27 [16 v7 c+32.r64 v6 c+32.r64 ]
m27

[8 v7 g32.r64 v6 g32.r64 ] t58
[4 v7 a32.r64 v6 a32.r64 ] t78
[2 v7 b32.r64 v6 b32.r64 ]
>f+64b64>e64g+64a16>c+16a16 t58

% true ending
v5 >e4 v3 e4 v2 e4&e16 r64 v6 <<c64<b64a+64a64g+16.&g+64 t42

v6 o1 g4 v5 g4 v6 f+4 t37 v5 f+4
v6 a4 v5 a4 v6 t32 g4 v5 g4
v6 >c4 v5 c4 v6 <a+4 t48 v5 a+4
v6 >c+4 t58 v5 c+4 v6 t78 <b4 v5 b4
v6 e2 v4 e2
v2 e2 v1 e1&e2.

r4

%~~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL D %~~~~~~~~~~~~~~~~~~~~~~~~~~%

@ t42 r4

% introduction & theme A1 / A2 (3/4)
[30 r1 ] r2.

% theme A & B (3/4)
[22 m8 ]
r1r1

% theme B1 (4/4)
[7 m14 ] m17
[6 m14 ] m20
d+64d+16.&r64r2.r8
e4e4e8d8d+64d+32.d+16d+16e16

% theme B2 (4/4)
```

```
[7 m14 ] m17
[6 m14 ] m20 m17

% theme C (4/4)
[7 m14 ] m17 m14 m14

% ending melody (3/4)
[12 r1 ] r2 t58 r2. t78 r2.

% C#maj13, Amaj13 (4/4)
t50 [6 r1 ]
[3 d16c+eeecc+de32ec+16ded+64d+32.c16de ]
e16e64eeec16c+eded+64d+32.e16cded+64d+16.&r64d+16d+

% Gmaj13#11/A, Amaj13#11, A#maj13#11/A & Dmaj9, Fmaj9 (6/8)
t34 [7 m25 d+64d+16.r64e8e16ed+64d+16.r64c+8d+ ]
m25
t42 d+64d+16.r64e32e32d+16d+16e16d+64d+16.r64c+16d+32d+32d+64d+16.r64

% theme E
[3 m26 d+64d+32.c+16d16d+64d+32.e16c16d16d+16e32e32d+16d16e16d+64d+32.c16d16d+16 ]
m26

% nothing else
[9 r1 ] r4

r4

%~~~~~~~~~~~~~~~~~~~~~~~~~%   MACRO   %~~~~~~~~~~~~~~~~~~~~~~~~~%

% macro #01 - channel B: arp Csus4 #1 (main riff) (3/4)
@ o5 v6 e64 v5 e32. v4 <b64 v3 b32. v6 >a64 v5 a32. v4 e64 v3 e32. v6
  b64 v5 b32. v4 a64 v3 a32. v6 e64 v5 e32. v4 b64 v3 b32. v6 <a64 v5
  a32. v4 >e64 v3 e32. v6 <b64 v5 b32. v4 a64 v3 a32.

% macro #02 - channel A: theme a1 #1 (3/4)
@ o3 v6 b16r16 v4 b16 v6 b16>e8. v5 e8. v4 e8. v3 e8. v2 e8&e32 v6
  f+32e32c+32<b16 v4 b16 v6 >d16 v4 d16 v6 f+16 v4 f+16 v6 a8g+8. v5
  g+8.a64a+64 v6 b8&b32 v5 b8. v4 b8.&b32a+64a64 v6 g+16 v5 g+16 v6
  e16 v5 e16 v6 g+16 v5 g+16 v6 b16 v5 b16 v6 a8. v5 a8. v6 g+8. v5
  g+8. v6 f+8. v5 f+8. v6 <b8.>d64f+64g+8&g+32

% macro #03 - channel A: theme a1 #2 (3/4)
@ o4 v6 e8. v5 e8. v6 <b8. v5 b8. v4 b8. v3 b8. v2 b8

% macro #04 - channel A: theme a1 #3 (3/4)
@ o4 v6 e8. v5 e8. v4 e8. v3 e8. v2 e8. v1 e4&e16

% macro #05 - channel A: theme a2 (3/4)
@ o4 v6 e16r16 v4 e16 v6 e16b32>c+8&c+32 v5 c+8. v4 c+8. v3 c+8.r4 v6
  c+32e32 v5 e16 v6 <b16 v5 b16 v6 a16 v5 a16 v6 >c16 v5 c16 v6 <b8.
  v5 b8. v6 >e8. v5 e8. v4 e4 v6 f+8r16 v5 f+16 v6 g+8r16 v5 g+16 v6
  a4r16 v5 a16 v6 g+4r16 v5 g+16 v6 d+16e4d+32d32c+4r16 v5 c+16 v6 <
  b4&b16 v5 b4&b16 v4 b4&b16 v3 b4&b16 v2 b4

% macro #06 - channel C: bass pedal C #1 (3/4)
@ o1 v6 e16 v5 e16r4 v6 e16 v5 e16r8 v6 e16 v5 e16

% macro #07 - channel C: bass pedal G (3/4)
@ o1 v6 b16 v5 b16r4 v6 b16 v5 b16r8 v6 b16 v5 b16

% macro #08 - channel D: kit pattern a (3/4)
@ d8ec+ded+64d+16.&r64

% macro #09 - channel B: arp Cmaj (interleaved with macro #01) (3/4)
@ o4 v5 [3e64<g+64b64>]
  v5 >a32. v4 e64 v3 e32. v6 b64 v5 b32. v4 a64 v3 a32.<
  v5 [3e64<g+64b64>]
  v5 a32. v4 >e64 v3 e32. v6 <b64 v5 b32. v4 a64 v3 a32. v6 >e64 v5
  e32. v4 <b64 v3 b32.
  v5 [3e64<g+64b64>]
  v5 >b32. v4 a64 v3 a32.<
  v5 [3e64<g+64b64>] v5 a32. v4 >e64 v3 e32. v6 <b64 v5 b32. v4 a64
  v3 a32.

% macro #10 - channel C: bass pedal F (3/4)
@ o1 v7 a16 v6 a16 v7 >>e16r8.<<a16 v6 a16 v7 >>e16r16<<a16 v6 a16
```

```
% macro #11 - channel C: bass pedal C #2 (3/4)
@ o1 v7 e16 v6 e16 v7 >>e16r8.<<e16 v6 e16 v7 >>e16r16<<e16 v6 e16

% macro #12 - channel B: arp Cmaj13 transition (3/4)
@ v7 [2 o3 a64>c+64e64g+64b64>c+64e64a64 v6 ]
  v5 [2 o3 a64>c+64e64g+64b64>c+64e64a64 v4 ]
  v3 [2 o3 a64>c+64e64g+64b64>c+64e64a64 v2 ]
  v3 [2 o5 e64r64<b64r64f+64r64<a64r64 v4 ]
  v5 [2 o5 e64r64<b64r64f+64r64<a64r64 v6 ]
  v7 [2 o5 e64r64<b64r64f+64r64<a64r64 v8 ]

  v4 >>e64c+64<b64 v5 a64g+64f+64 v6 e64c+64<b64
  v7 a64g+64f+64 v8 e64c+64<b64a64

% macro #13 - channel C: bass theme b #1 (4/4)
@ o2 v7 e16 v6 e16r16 v8 >>e64r32.<e64r16.r64 v7 <e16 v6 e16r8. v8
  >>e64r32.<e64r8.r32.
  v7 <d+16 v6 d+16r16 v8 >>e64r32.<e64r16.r64 v7 <d+16 v6 d+16r8. v8
  >>e64r32.<e64r8.r32.
  v7 <d16 v6 d16r16 v8 >>e64r32.<e64r16.r64 v7 <d16 v6 d16r8. v8
  >>e64r32.<e64r8.r32.
  v7 <c+16 v6 c+16r16 v8 >>e64r32.<e64r16.r64 v7 <c+16 v6 c+16r8. v8
  >>e64r32.<e64r8.r32.
  v7 <c16 v6 c16r16 v8 >>e64r32.<e64r16.r64 v7 <c16 v6 c16r8. v8
  >>e64r32.<e64r8.r32.
  v7 <<b16 v6 b16r16 v8 o4 e64r32.<e64r16.r64 v7 <<b16 v6 b16r8. v8
  o4 e64r32.<e64r8.r32.

% macro #14 - channel D: kit b (4/4)
@ d8e32eeed+8c+edd+64d+16.&r64c8

% macro #15 - channel A: theme b1 #1 (4/4)
@ o2 v6 b16r8b16>e4 v4 e4 v3 e8 v6 e16d+16
  e16<b16>e16g+16e16f+8. v4 f+4 v3 f+4
  v6 <b16r16 v3 b16 v6 b16>g+4 v4 g+4 v3 g+8 v6 g+16g16
  g+16e16g+16b16a4 v4 a4 v3 a4
  v6 c+16r16 v3 c+16 v6 c+16a8. v4 a8.r8 v6 a16r16g+32a32g+16
  f+8a8f+32g+16.r16 v3 g+16 v6 d+32e16.r16 v3 e16 v6 <b8r16 v3 b16
  v6 b16r16 v3 b16 v6 b16

% macro #16 - channel C: bass theme b #2 (4/4)
@ o1 v7 a+16 v6 a+16r16 v8 o4 e64r32.<e64r16.r64 v7 <<a+16 v6 a+16r8.
  v8 o4e64r32.<e64r8.r32. v7 <<b16 v6 b16r16 v8 o4 e64r32.<e64r16.r64
  v7 <<b16 v6 b16 v8 o4 c+64c64r32<a64
  g+64r32a64g+64r32>e64r32.<e64r16.r64a64g+64g64f+64f64e64r32

% macro #17 - channel D: kit b fill #1 (4/4)
@ d8e32eeed+8c+d+16d+d8d+64d+16.&r64d+8

% macro #18 - channel A: theme b ending (4/4)
@ o3 v6 c+8r16 v3 c+16 v6 e8r16 v3 e16 v6 e32f+16.r32 v3 e32f+16 v6
  f+32g+16.r32 v3 f+32g+16 v6 e32f+32e8. v5 e4 v4 e4 v3 e4

% macro #19 - channel B: arp Csus4 #1 stub (3/4)
@ o5 v6 e64 v5 e32. v4 <b64 v3 b32. v6 >a64 v5 a32. v4 e64 v3 e32. v6
  b64 v5 b32. v4 a64 v3 a32. v6 e64 v5 e32. v4 b64 v3 b32.

% macro #20 - channel D: kit b fill #2 (4/4)
@ d4dde32eeec8

% macro #21 - channel A: theme b2 #1 (4/4)
@ [16 o3 v5 e64g+64 v7 b64 ]
  [5 o3 v5 e64g+64 v7 >e64 ] r64
  [16 o3 v5 d+64g+64 v7 b64 ]
  [5 v5 d+64g64 v7 a+64 ] r64

% macro #22 - channel A: arp theme c C#maj7 (4/4)
@ o3 [2 [3 g+64e64c+64 ] r8.r32. [3 g+64e64c+64 ] r4.r16.r64 ]

% macro #23 - channel B: arp theme d Dmaj9 (6/8)
@ [4 v6 o5 c+32 v4 <<f32 v6 >f32 v4 >c+32 v6 <g+32 v4 f32 v6 <f+32
v4 >g+32 v6 <a+32 v4 f+32 v6 f32 v4 a+32 v6 ]

% macro #24 - channel B: arp theme d Fmaj9 (6/8)
@ [2 o5 e32 v4 <<e32 v6 >g+32 v4 >e32 v6 <b32 v4 g+32 v6 <a32 v4 >b32 v6
c+32 v4 <a32 v6 e32 v4 >c+32 v6 ]

% macro #25 - channel D: kit d fragment (6/8)
```

```
@ d8c+eeec+d+64d+16.r64e8c+edd+64d+16.r64d8c+eeed

% macro #26 - channel D: kit e pattern (4/4)
@ [5 d16c+16e16e16d+64d+32.c16c+16d+16e32e32c16d16e16d+64d+32.c16e16d+16 ]

% macro #27 - channel C: theme e bass rise (4/4)
@ o1 [8 v7 c+32.r64 v6 c+32.r64 ]
  [8 v7 d+32.r64 v6 d+32.r64 ]
  [8 v7 e32.r64 v6 e32.r64 ]
  [8 v7 f+32.r64 v6 f+32.r64 ]

% macro #28 - channel C: theme e bass pedal #1 (4/4)
@ o1 [16 v7 g+32.r64 v6 g+32.r64 ]

% macro #29 - channel A: theme e arp #1 (4/4)
@ v4 [10 o4 c+32<g+32f+32 ] g+32>c+32
[10 o4 d+32<a+32g+32 ] a+32>d+32
[10 o4 d32<a32g32 ] a32>d32
[10 o4 e32<b32a32 ] a32>b32

% macro #30 - channel A: theme e arp #2 (4/4)
@ v4 o4 [10 c+32g+32d+32 ] c+32d+32

% macro #31 - channel A: main theme e #1 (4/4)
@ o3 v7 g+16r8g+16 v8 >c+32 v7 c+8.&c+32 v4 c+4 v3 c+8 v7 >c+64<c+32.>c64<c32.
>c+64<c+32.g+64<g+32.>>c+64<c+32.>f64<f32.>c+64<c+32. v8 d+32 v7 d+8&d+32
v4 d+8. v3 d+4&d+16
v7 <a+16r16 v3 a+16 v7 a+16 v8 >e32 v7 e8.&e32 v4 e4 v3 e8 v7 >e64<e32.>d+64<d+32.
>e64<e32.b64<b32.>>e64<e32.>g+64<g+32. v8 f+32 v7 f+8.&f+32 v4 f+8. v3 f+8.
v7 >f+64<f+32.>f64<f32.
>f+64<f+32.>c+64<c+32.>f+64<f+32.>a+64<a+32.

% macro #32 - channel A: true ending held note (4/4)
@ o5 v7 e2 v6 e2 v5 e2 v4 e2 v3 e2 v2 e2

% macro #33 - channel B: theme e arp #1 (4/4)
@ [3 o5 v6 c+32 v4 <c32 v6 f32 v4 >c+32 v6 <g+32 v4 f32 v6 <g+32 v4 >g+32 v6 c32 v4 <g+32 ]
  v6 >c+32g+32

% macro #34 - channel B: theme e arp #2 (4/4)
@ [3 o5 v6 d+32 v4 <c32 v6 g32 v4 >d+32 v6 <a+32 v4 g32 v6 <g+32 v4 >a+32 v6 c32 v4 <g+32 ]
  v6 >d+32a+32

% macro #35 - channel B: theme e arp #3 (4/4)
@ [3 o5 v6 e32 v4 <d+32 v6 g+32 v4 >e32 v6 <b32 v4 g+32 v6 <b32 v4 >b32 v6 d+32 v4 <b32 ]
  v6 >d+32b32
  [3 o5 v6 f+32 v4 <g+32 v6 a+32 v4 >f+32 v6 c+32 v4 <a+32 v6 c+32 v4 >c+32 v6 <g+32 v4 c+32 ]
  v6 >g+32c+32

% macro #36 - channel B: theme e arp #4 (4/4)
@ [5 o3 f64>c64a64<a64c64>f64 ]
  f64>c64

% macro #37 - channel B: arp Csus4 echo (3/4)
@ o4 v6 a64 v5 a32. v4 >e64 v3 e32. v6 <b64 v5 b32. v4 a64 v3 a32.
  v4 a64 v3 a32. v2 >e64 v1 e32. v4 <b64 v3 b32. v2 a64 v1 a32.
  v3 a64 v2 a32. v1 >c16 v4 <b64 v2 b32. v1 c16

%=====================================================================%
```

6.4 jupiter.mmml

```
%====================================================================%
% TITLE      : Jupiter
% COMPOSER   : Blake 'PROTODOME' Troise
% PROGRAMMER : Blake 'PROTODOME' Troise
% DATE       : 3rd August 2018
% NOTES      : 'Jupiter' is a suite of minimalist, 1-bit sonic
%              textures representing the environments, topographies
%              and unique characteristics of the Jovian moons.
%
%              Written for three, pin-pulse mixed, 1-bit pulse waves.
%
%              Through the use of phase shifting, each piece is
%              extrapolated from a few, simple lines of musical
%              material, producing kaleidoscopic, aural moiré
%              patterns.
%
%              When compiling, make sure the SYNTH_TYPE definition
%              is commented out.
%====================================================================%

%~~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL A %~~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% ganymede
t255                                % set section I (intro) tempo
o1 [9 [5 m3 ] t1 ]                  % section I
t180                                % set section II (main) tempo
[4 m1 ] [4 m2 ]                     % section II
[5 m3 ]                             % section III (sync w/ channel c)

% europa
t34                                 % set section I tempo
o4 [16 m4 ] [8 m5 ]                 % section I
r4.                                 % sync w/ channel c
[4 r1 ] t68 [4 r ] t136 [2 r ]      % transition (slow tempo)
[4 m5 ] [2 m4 ]                     % section II
r16.                                % sync w/ channel c
v2 c1                               % transition to io

% io
t110                                % set global tempo
m6                                  % section I
[64 m7]                             % section II
o1 v8 c1&c64.                       % sync w/ channel c + distortion
[4 r64. v8 c1 v5 g1 ]               % section III
t220                                % slow tempo
v6 [2 [2 m8 ] v8 ]                  % section VI

% jupiter
v8 c1 v7 c v6 c v5 c
v4 c v3 c v2 c v1 c

[255 r1 ]                           % shh

%~~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL B %~~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% ganymede
o4 [9 r1 [4 m3 ]]                   % section I
[8 r4 m1 ]                          % section II
[3 m3 ]                             % section III (sync w/ channel c)

% europa
[16 r128 m4 ] [8 r m5 ]             % section I
r8.                                 % sync w/ channel c
[5 m5 ]                             % transition
[4 r128 m5 ] [2 r m4 ]              % section II
r32.                                % sync w/ channel c
r1                                  % transition to io

% io
r128                                % offset material from channel a
[3 r1 ] m6                          % section I
```

```
[50 m7 r128]                        % section II
o1 v8 c#1&c#8                       % sync w/ channel c + distortion
[4 v8 c#1 v5 g#1 ]                  % section III
r16.                                % sync w/ channel a
o2 v6 [2 [2 r128 m8 ] v8 ]          % section VI

% jupiter
v8 g1 v7 g v6 g v5 g
v4 g v3 g v2 g v1 g2.r8.r32.

[255 r1 ]                           % shh

%~~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL C %~~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4

% ganymede
o3 [9 r1r m3 m1 r8 ]                % section I
[4 r2 m1 ] [4 r m2 ]                % section II
t255                                % set section III (outro) tempo
m3                                  % section III

% europa
[16 r64 m4 ] [8 r m5 ]              % section I
[10 r1 ]                            % transition
[4 r64 m5 ] [2 r m4 ]               % section II
r1                                  % transition to io

% io
o4 r64.                             % offset material from channel a
[5 r1 ] m6                          % section I
[5 v8 m8 v6 m8 v4 m8 v2 m8 ]        % section II & jupiter section I
r16.                                % sync w/ channel a
o3 v8 [4 m7 ]                       % section I

% jupiter
v8 e1 v7 e v6 e v5 e                % end of io & section I
v4 e v3 e v2 e v1 e
[4 r1]                              % sync w/ channels a & b

[255 r1 ]                           % shh

%~~~~~~~~~~~~~~~~~~~~~~~~~~% CHANNEL D %~~~~~~~~~~~~~~~~~~~~~~~~~~%

@ r4 % nothing to do

%~~~~~~~~~~~~~~~~~~~~~~~~~~%   MACRO   %~~~~~~~~~~~~~~~~~~~~~~~~~~%

% m1  : ganymede eminor, duration: r1 r2 r4.
@ [8v4e128v5e]v5e8v4ev3ev2e v4g8v5[8v3g128v4g]g8v3gv2g
  v3d32v4dv5dv6dv5d8v4dv3dv2d

% m2  : ganymede cadd9, duration: r1 r2 r4.
@ [8v4c128v5c]v5c8v4cv3cv2c v4d8v5[8v3d128v4d]d8v3dv2d
  v3e32v4ev5ev6ev5e8v4ev3ev2e

% m3  : ganymede c pulsing pedal, duration: r1
@ [64v1c128v3c]

% m4  : europa cmaj13, duration: r1 r1
@ [16 v3c64v4egv6bv5dv4av3ev2g ]

% m5  : europa g#maj13, duration: r1 r1
@ [16 v3g#64v4a#cv6d#v5gv4a#v3cv2g# ]

% m6  : io c#minor7 #1
@ v2 [192 c#128df#a ] v3 [192 c#df#a ] v4 [192 c#df#a ]
  v6 [192 c#df#a ] v8 [192 c#df#a ]

% m7  : io c#minor7 #2
@ v3 [2 c#128df#a ] v4 [2 c#df#a ] v6 [2 c#df#a ] v8 [2 c#df#a ]

% m8  : io c#minor7 #3
@ [32 o5 f#128 o4 f# o3 f# o2 f# ]

%====================================================================%
```

6.5 bitbeat.c

```
/*H*********************************************************
*  FILENAME:       bitbeat.c
*  DESCRIPTION:    Bitbeat Suite
*
*  NOTES:          A selection of bitbeat pieces squeezed
*                  into 915 bytes. To cycle through pieces,
*                  simply switch the Attiny13 on and off!
*
*  AUTHOR:         Blake Troise
*  PLATFORM:       Attiny13 4MHz
*  DATE:           17th August 2019
*  SIZE:           915 bytes
*********************************************************H*/

#include <avr/io.h>
#include <avr/eeprom.h>

// variables
uint8_t current_piece;
uint8_t out1;
uint8_t out2;
uint8_t out3;

int main(void){
  // configure output for mono
  DDRB = 0b00000001;

  // read song position from EEPROM
  uint8_t current_piece = eeprom_read_byte((uint8_t*)0);

  // change track for next startup
  if(current_piece < 3)
    eeprom_write_byte(0,current_piece + 1);
  else
    eeprom_write_byte(0,0);

  while(1){
    // millipede call centre
    if(current_piece == 0){
      for(uint16_t l = 150 ;; l+=50)
      {
        for(uint16_t t = 0; t < 65535; t++)
        {
          PORTB = (t >> PORTB | PORTB >> 1) ^ 1;
          for(uint16_t i = 0; i < (l & t); i++) asm("nop");
        }
      }
    }

    // modem exorcism pt.1
    if(current_piece == 1){
      for(uint8_t v = 1; v < 255; v++){
        for(uint16_t t = 0; t < 65535; t++){
          out2 = (t << out2) | (out2 >> 1);
          PORTB = out2;
          out3 = (t << out3) | t;
          PORTB = out3;
          out1 = (t >> out1) | (out1 >> v);
          PORTB = out1;
          for(uint16_t i = 0; i < (250 ^ PORTB); i++) asm("nop");
        }
      }
    }

    // fax attack
    if(current_piece == 2){
      for(uint16_t v = 200 ;; v+=16){
        for(uint16_t t = 0; t < 2300; t++){
          PORTB = (t << PORTB) | (t >> PORTB);
          PORTB = (t << PORTB) ^ (t << v);
          PORTB = (t >> v) ^ (PORTB >> t);
          for(uint16_t i = 0; i < ((v & (t << PORTB)) + (132 & (t << PORTB))); i++) asm("nop");
        }
      }
    }
```

```
    // tiny djent
    if(current_piece == 3){
      for(uint16_t l = 550; l < 1150; l+=50){
        for(uint16_t t = 0; t < 32767; t++){
          out2 = (t << out2) | (out2 >> l) ^ 1;
          PORTB = out2;
          out1 = (t >> out1) | (out1 >> l) ^ 1;
          PORTB = out1;
          out1 = (t >> out1 | out1 >> 1) ^ 1;
          PORTB = out1;
          for(uint16_t i = 0; i < (l & t); i++) asm("nop");
        }
      }
      for(uint16_t l = 1 ; l < 7; l++){
        for(uint16_t t = 0; t < 65535; t++){
          out3 = (t << out3) | (t >> l);
          PORTB = out3;
          out2 = (t << out2) | (out2 >> l);
          PORTB = out2;
          out1 = (t >> out1) | (out1 >> l);
          PORTB = out1;
          for(uint16_t i = 0; i < (150 ^ PORTB); i++) asm("nop");
        }
      }
    }
  }
}
```

7 References

[1] A. Carlsson, "DEMOSCENE," *ChipFlip*, 2017. [Online]. Available: https://chipflip.wordpress.com/demoscene/ (Accessed 03/05/2017).

[2] utz, "Sizecoding," *Ancient Wonderland*. [Online]. Available: https://irrlichtproject.blogspot.co.uk/search/label/sizecoding (Accessed 05/04/2017).

[3] M. Reunanen, "How Those Crackers Became Us Demosceners," April 2014. [Online]. Available: http://widerscreen.fi/numerot/2014-1-2/crackers-became-us-demosceners/ (Accessed 03/05/2017).

[4] "Revision 2017: Results," *Revision*, April. [Online]. Available: https://2017.revision-party.net/history/2017 (Accessed 03/05/2017).

[5] "Composition Techniques Specific to Chiptune?" *FamiTracker Forum*, September 2012. [Online]. Available: http://famitracker.com/forum/posts.php?id=3908 (Accessed 04/05/2017).

[6] Vhiiula/Analogik, "Techniques of Chipping - A detailed tutorial on how to create chiptunes," *Milkytracker Documents*. [Online]. Available: http://milkytracker.titandemo.org/docs/Vhiiula-TechniquesOfChipping.txt (Accessed 04/05/2017).

[7] L. Paul, *For the Love of Chiptune*, ser. Oxford Handbooks, K. Collins, B. Kapralos, and H. Tessler, Eds. Oxford University Press, 2014.

[8] C. Hopkins, "Chiptune Music: An Exploration of Compositional Techniques Found in Sunsoft Games for the Nintendo Entertainment System and Famicom from 1988 - 1992," Ph.D. dissertation, Five Towns College, March 2015.

[9] T. Perich, "1-Bit Symphony, by Tristan Perich," *Tristan Perich*, March 2015. [Online]. Available: https://tristanperich.bandcamp.com/album/1-bit-symphony (Accessed 28/12/2018).

[10] Atmel, "ATtiny25/V / ATtiny45/V / ATtiny85/V," *Atmel 8-bit AVR Microcontroller with 2/4/8K Bytes In-System Programmable Flash*. [Online]. Available: http://www.atmel.com/images/atmel-2586-avr-8-bit-microcontroller-attiny25-attiny45-attiny85_datasheet.pdf (Accessed 07/04/2017).

[11] R. Burkey, "Virtual AGC —AGS —LVDC —Gemini," *Programmer's Manual Block 2 AGC Assembly Language*, July 2018. [Online]. Available: https://www.ibiblio.org/apollo/assembly_language_manual.html#The_Interpreter_vs._the_CPU

[12] J. Lazzaro and J. Wawrzynek, "MPEG-4 Structured Audio: Developer Tools," *mp4-sa*, 2000. [Online]. Available: https://john-lazzaro.github.io/sa/index.html (Accessed 26/09/2018).

[13] YERZMYEY, "1-BIT CHIPTUNES / BEEPER MUSIC," *CMO.org*, 2013. [Online]. Available: http://chipmusic.org/forums/topic/12566/1bit-chiptunes-beeper-music/ (Accessed 09/04/2017).

[14] A. Silvast, M. Reunanen, and G. Albert, "Demoscene Research." [Online]. Available: http://www.kameli.net/demoresearch2/ (Accessed 04/04/2017).

[15] "Ada Lovelace," *The Babbage Engine*, Computer History Museum. [Online]. Available: http://www.computerhistory.org/babbage/adalovelace/ (Accessed 05/04/2017).

[16] Jack Copeland and Jason Long, "Christmas carols from turing's computer," *Sound and vision blog*, December 2017. [Online]. Available: https://blogs.bl.uk/sound-and-vision/2017/12/christmas-carols-from-turings-computer.html (Accessed 02/06/2019).

[17] utz, "Computer Music in 1949?" *Ancient Wonderland*, irrlicht project, November 2015. [Online]. Available: http://irrlichtproject.blogspot.co.uk/2015/11/computer-music-in-1949.html (Accessed 05/04/2017).

[18] A. Carlsson, "TIMELINE," *ChipFlip*, 2017. [Online]. Available: https://chipflip.wordpress.com/timeline/ (Accessed 03/05/2017).

[19] J. Fildes, "'Oldest' computer music unveiled," *Technology*, BBC News, June 2008. [Online]. Available: http://news.bbc.co.uk/1/hi/technology/7458479.stm (Accessed 05/04/2017).

[20] "First digital music made in Manchester," *Technology*, The University of Manchester, June 2008. [Online]. Available: http://www.manchester.ac.uk/discover/news/first-digital-music-made-in-manchester (Accessed 05/04/2017).

[21] K. Kleiman, "Singing BINAC - 1948," *CYHIST Community Memory: Discussion list on the History of Cyberspace*, November 1997. [Online]. Available: https://groups.yahoo.com/neo/groups/cyhist/conversations/messages/1271 (Accessed 05/04/2017).

[22] A. Turing, "Programmers' Handbook for Manchester Electronic Computer Mark II," *The Manchester Computer*, AlanTuring.org, p. 24, 1950–1952. [Online]. Available: http://www.alanturing.net/turing_archive/archive/m/m01/M01-030.html (Accessed 06/04/2017).

[23] Jack Copeland and Jason Long, "Restoring the first recording of computer music," *Sound and vision blog*, September 2016. [Online]. Available: https://blogs.bl.uk/sound-and-vision/2016/09/restoring-the-first-recording-of-computer-music.html (Accessed 02/06/2019).

[24] "Nellie: School Computer," *Tomorrow's World, Series 4*, BBC Broadcasting Service, February 1969. [Online]. Available: http://www.bbc.co.uk/programmes/p0154hns (Accessed 06/04/2017).

[25] D. Hartley, "EDSAC 1 and after - a compilation of personal reminiscences," *EDSAC 99*, 1999. [Online]. Available: https://www.cl.cam.ac.uk/events/EDSAC99/reminiscences/ (Accessed 06/04/2017).

[26] D. Sordillo, "Music Playing on the PDP-6," *Project MAC*, August 1966. [Online]. Available: ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-107.pdf (Accessed 06/04/2017).

[27] P. Doornbusch, "Computer sound synthesis in 1951: The music of csirac," *Computer Music Journal*, vol. 28, 03 2004.

[28] N. Hardy, "Music," *Stories*, 2005. [Online]. Available: http://www.cap-lore.com/stories/music.html (Accessed 06/04/2017).

[29] Zuse. (1970) Programm 47. [Online]. Available: http://www.sol20.org/manuals/music.pdf (Accessed 30/04/2019).

[30] G. Sundqvist, "D21 - In Memoriam - D22," Vinyl, EP, Sweden: KDA –RM 5301, 2016.

[31] T. Van Keuren, "Ebb Tide played by 1970 Univac Computer (No Sound Card)," January 2015. [Online]. Available: https://www.youtube.com/watch?v=X6F7qwa5TZg (Accessed 06/04/2017).

[32] S. T. Corporation, "The Music System," *Music System User's Manual*, 1977. [Online]. Available: http://www.sol20.org/manuals/music.pdf (Accessed 06/04/2017).

[33] M. Fritsch, *"History of Video Game Music"*, pp. 11—41, 08 2012.

[34] J. Latimer, "Hit It, Maestro!" *Compute! Magazine*, April 1990. [Online]. Available: https://web.archive.org/web/20140906061745/http://www.joeylatimer.com/pdf/Compute!%20April%201990%20PC%20Sound%20Gets%20Serious%20by%20Joey%20Latimer.pdf (Accessed 29/03/2019).

[35] Joakim Ögren. (1997) The Hardware Book. [Online]. Available: http://www.acc.umu.se/~stric/tmp/hwb13pdf/hwbook.pdf#page=290 (Accessed 30/04/2019).

[36] S. Vickers, *ZX Spectrum BASIC Programming*, R. Bradbeer, Ed. Sinclair Research, 1982.

[37] id Software, "Doom," MS-DOS, 1993.

[38] Brøderbund, "Prince of persia," MS-DOS, 1989.

[39] Maxis, "Simcity 2000," MS-DOS, 1993.

[40] Incentive Software, "Total eclipse," MS-DOS, 1988.

[41] Access Software Inc., *Crime Wave Instruction Manual*. Holford, Birmingham, United Kingdom: US. Gold Ltd, 1990.

[42] M. Fritsch, *"History of Video Game Music"*, pp. 11—41, 08 2012.

[43] G. Herman, *Micromusic for the Commodore 64 and BBC Computer*, pp. 22–23. London: PAPERMAC, 1985.

[44] N. Houston, "Music Made on Game Boys Is a Much Bigger Deal Than You'd Think," *VICE vs Video Games*, November 2014. [Online]. Available: https://www.vice.com/en_uk/article/chipzels-complete-history-of-chiptune-939 (Accessed 06/04/2017).

[45] M. Wright, "New Horizons for Microcomputer Music," *The Best of Creative Computing*, 1980. [Online]. Available: http://www.atariarchives.org/bcc3/showpage.php?page=82 (Accessed 06/04/2017).

[46] M. Z, Arnauld, and Kyle, "PSG," *IntelliWiki*, September 2011. [Online]. Available: http://ozzed.net/how-to-make-8-bit-music.shtml (Accessed 15/05/2017).

[47] G. Instruments, "AY-3-8910/8912 Programmable Sound Generator Data Manual," February 1979. [Online]. Available: http://dev-docs.atariforge.org/files/GI_AY-3-8910_Feb-1979.pdf (Accessed 15/05/2017).

[48] B. Troise, "Compositional Strategies For Programmable Sound Generators With Limited Polyphony," *Ludomusicology*, July 2015. [Online]. Available: http://www.ludomusicology.org/2015/07/16/compositional-strategies-for-programmable-sound-generators-with-limited-polyphony/ (Accessed 15/05/2017).

[49] A. Carlsson, "Post-Chiptune is All About Culture?" *ChipFlip*, 2016. [Online]. Available: https://chipflip.wordpress.com/2016/11/29/post-chiptune-is-all-about-culture/ (Accessed 25/12/2018).

[50] B. Eno, *A Year With Swollen Appendices*. London: Faber and Faber, 1996.

[51] A. Carlsson, "CHIPMUSIC," *ChipFlip*, 2007. [Online]. Available: https://chipflip.wordpress.com/chipmusic/ (Accessed 03/05/2017).

[52] S. Tomczak, "Authenticity and Emulation: Chiptune in the Early Twenty-First Century," *Conference Paper at the International Computer Music Conference*, August 2008. [Online]. Available: https://quod.lib.umich.edu/cgi/p/pod/dod-idx/authenticity-and-emulation-chiptune-in-the-early-twenty.pdf?c=icmc;idno=bbp2372.2008.035 (Accessed 03/05/2017).

[53] B. Hood, "WHAT *IS* CHIPTUNE?" *The ChipWin Blog*, May 2016. [Online]. Available: http://chiptuneswin.com/blog/what-is-chiptune/ (Accessed 03/05/2017).

[54] "Chiptune," *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Chiptune (Accessed 03/05/2017).

[55] L. Ohanesian, "What, Exactly, is 8-Bit Music?" *LA Weekly*, August 2011. [Online]. Available: http://www.laweekly.com/music/what-exactly-is-8-bit-music-2409754 (Accessed 03/05/2017).

[56] G. Lynch, "From 8-bit to Chiptune: the music that changed gaming forever," *techradar.*, March 2017. [Online]. Available: http://www.techradar.com/news/8-bit-music-the-soundtrack-to-a-gaming-revolution-that-resonates-today (Accessed 03/05/2017).

[57] J. List, "Sega Genesis Chiptunes Player Uses Original Chips," *HACKADAY*, February 2017. [Online]. Available: http://hackaday.com/2017/02/17/sega-genesis-chiptunes-player-uses-original-chips/ (Accessed 03/05/2017).

[58] G. Wittel, "SEGA Genesis Specs," *dEX*, 2000. [Online]. Available: http://dextremes.com/genesis/gen-spec.html (Accessed 07/05/2017).

[59] "Sega Mega Drive," *SEGA Retro*, May 2017. [Online]. Available: http://segaretro.org/Sega_Mega_Drive (Accessed 07/05/2017).

[60] Yamaha, "Yamaha TX81Z FM Tone Generator Owner's Manual," Hamamatsu, Japan, May 1987.

[61] K. Driscoll and D. Joshua, "Endless loop: A brief history of chiptunes," *Transformative Works and Cultures*, vol. 2, 2009. [Online]. Available: http://journal.transformativeworks.org/index.php/twc/article/view/96/94 (Accessed 04/04/2017).

[62] L. Akesson, "Elements of Chip Music," *Revision Party*, linusakesson.net, 2011. [Online]. Available: http://www.linusakesson.net/music/elements/ (Accessed 29/03/2017).

[63] "How to Make 8 Bit Music?" *gamedev.net*, May 2011. [Online]. Available: https://www.gamedev.net/topic/602786-how-to-make-8-bit-music/ (Accessed 12/05/2017).

[64] J. Allen, "How To Make 8-Bit Music: An Introduction To FamiTracker," *Synthtopia*, May 2015. [Online]. Available: http://www.synthtopia.com/content/2015/05/01/how-to-make-8-bit-music-an-introduction-to-famitracker/ (Accessed 12/05/2017).

[65] Ozzed, "How to Make 8-bit Music," *ozzed.net*, May 2015. [Online]. Available: http://ozzed.net/how-to-make-8-bit-music.shtml (Accessed 12/05/2017).

[66] S. Sandhu, "Tristan Perich: he's a one-bit wonder," *The Telegraph*, November 2010. [Online]. Available: https://www.telegraph.co.uk/culture/music/rockandpopfeatures/8163589/Tristan-Perich-hes-a-one-bit-wonder.html (Accessed 28/12/2018).

[67] A. Carlsson, "What's Chipmusic in 2015?" *ChipFlip*, 2015. [Online]. Available: https://chipflip.wordpress.com/2015/11/13/whats-chipmusic-in-2015/ (Accessed 25/12/2018).

[68] J. Nisperos, "Moe Moe Kyunstep," *Bandcamp*, October 2012. [Online]. Available: https://chibitech.bandcamp.com/album/moe-moe-kyunstep (Accessed 22/12/2018).

[69] "Nintendo game boy (ch4033)," *Centre For Computing History*, Apr 2020. [Online]. Available: https://www.computinghistory.org.uk/det/4033/Nintendo-Game-Boy/ (Accessed 07/04/2020).

[70] P. Tiefenbacher, "Parametric Music Box," *Thingiverse*, March 2013. [Online]. Available: https://www.thingiverse.com/thing:53235 (Accessed 07/04/2020).

[71] P. Zadrożniak, "The Floppotron," July 2016. [Online]. Available: https://www.youtube.com/watch?v=Oym7B7YidKs (Accessed 07/04/2020).

[72] Wintergatan, "Wintergatan - Marble Machine (music instrument using 2000 marbles)," March 2016. [Online]. Available: https://www.youtube.com/watch?v=X6F7qwa5TZg (Accessed 07/04/2020).

[73] Robkta, "Switchtunes," *GameChops*, September 2019. [Online]. Available: http://gamechops.com/switchtunes/ (Accessed 08/04/2020).

[74] T. Farah, "Anamanaguchi: "Chiptune Isn't Really A Genre, It's A Medium"," *Phoenix New Times*, May 2016. [Online]. Available: https://www.phoenixnewtimes.com/music/anamanaguchi-chiptune-isnt-really-a-genre-its-a-medium-6609601

[75] A. Yabsley, "The Sound of Playing: A Study into the Music and Culture of Chiptunes," Ph.D. dissertation, Queensland Conservatorium Griffith University, October 2007, chiptune isn't a genre like rock for instance, where the characteristic lines of the genre is drawn from the arrangement and construction of the music, as well as the instruments. No, "chip" is the instrument itself, and with it (or should I say "them") you can make music from all kind of genres, like rock, pop, dnb [drum and bass], house or dub (para. 2).

[76] Vox, "The most feared song in jazz, explained," *Earworm*, November 2018. [Online]. Available: https://www.youtube.com/watch?v=62tIvfP9A2w (Accessed 22/12/2018).

[77] J. Nisperos, "Chibi-Tech - Moe Moe Kyunstep," *chipmusic.org*, October 2012. [Online]. Available: https://chipmusic.org/forums/topic/8899/chibitech-moe-moe-kyunstep/ (Accessed 22/12/2018).

[78] T. Perich, "1-Bit Symphony," *Physical Editions*, 2010. [Online]. Available: http://www.1bitsymphony.com/ (Accessed 09/04/2017).

[79] G. Oldham *et al.*, "Harmonics," *The Oxford Companion to Music*, Oxford University Press. [Online]. Available: http://www.oxfordmusiconline.com/subscriber/article/grove/music/50023 (Accessed 25/03/2017).

[80] E. Prestini, *The Evolution of Applied Harmonic Analysis: Models of the Real World.*, p. 62. Boston: Birkhäuser, 2004.

[81] "Phase," National Institute of Standards and Technology, September 2016. [Online]. Available: https://www.nist.gov/time-and-frequency-services/p (Accessed 25/03/2017).

[82] G. Herman, *Micromusic for the Commodore 64 and BBC Computer*, pp. 28–29. London: PAPERMAC, 1985.

[83] J. Borwick, "Frequency," *The Oxford Companion to Music*, Oxford University Press. [Online]. Available: http://www.oxfordmusiconline.com/subscriber/article/opr/t114/e2693 (Accessed 25/03/2017).

[84] "White Noise," *The Oxford Companion to Music*, Oxford University Press. [Online]. Available: http://www.oxfordmusiconline.com/subscriber/article/opr/t114/e8240 (Accessed 25/03/2017).

[85] H. Weixelbaum, "Game Boy Sound Comparison," *Game Boy Music*. [Online]. Available: http://www.herbertweixelbaum.com/comparison.htm (Accessed 25/03/2017).

[86] S. Lakawicz, "The Difference Between Pulse Waves and Square Waves," *Research in Game Music*, Classical Gaming, May 2012. [Online]. Available: https://classicalgaming.wordpress.com/2012/05/

15/research-in-game-music-the-difference-between-pulse-waves-and-square-waves/ (Accessed 24/03/2017).

[87] R. Middleton, *Know Your Square Wave and Pulse Generators*, pp. 21,29,49. H. W. Sams, 1965. [Online]. Available: https://books.google.co.uk/books?id=cCtTAAAAMAAJ

[88] K. C. Pohlmann, *Principles of Digital Audio*, 6th ed., p. 86. New York City: McGraw-Hill, 2011.

[89] L. Butler, "Waveforms Using The Cathode Ray Oscilloscope," *Waveform and Spectrum Analysis*, June 1989. [Online]. Available: http://users.tpg.com.au/users/ldbutler/Waveforms.htm (Accessed 26/03/2017).

[90] E. G. Louis, "Practical Techniques of Square-Wave Testing," *Radio & TV News*, RF Cafe, July 1957. [Online]. Available: http://www.rfcafe.com/references/radio-news/practical-techniques-square-wave-testing-july-1957-radio-tv-news.htm (Accessed 28/03/2017).

[91] Paul, "Chronos," *Crash*, vol. 41, p. 21, June 1987.

[92] Sweetwater, "Pink Noise Versus White Noise," *inSync*, August 2000. [Online]. Available: https://www.sweetwater.com/insync/pink-noise-versus-white-noise/ (Accessed 9/12/2018).

[93] D. Didier, "3x Osc," *FL Studio 20 Reference Manual*. [Online]. Available: http://www.image-line.com/support/flstudio_online_manual/html/plugins/3x%20OSC.htm (Accessed 5/12/2018).

[94] L. Erickson, "Nintendo Game Boy DMG-01: "Line Out Mod"," *GB Classic Audio Mod*, Low-Gain. [Online]. Available: http://lowgain-audio.com/GBclassicmod.htm (Accessed 29/03/2017).

[95] J. Kotlinski, "Game Boy Color ProSound Modification," Little Sound DJ. [Online]. Available: http://www.littlesounddj.com/lsd/prosound/ (Accessed 29/03/2017).

[96] R. Nave, "RC Low Pass Filter," *Electricity and Magnetism*, August 2010. [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/electric/filcap2.html (Accessed 07/04/2017).

[97] T. Oohashi, E. Nishina, M. Honda, Y. Yonekura, Y. Fuwamoto, N. Kawai, T. Maekawa, S. Nakamura, H. Fukuyama, and H. Shibasaki, "Inaudible High-Frequency Sounds Affect Brain Activity: Hypersonic Effect," *Journal of Neurophysiology*, vol. 83, no. 6, pp. 3548–3558, 2000. [Online]. Available: http://jn.physiology.org/content/83/6/3548

[98] S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, pp. 243–260. San Diego, CA, USA: California Technical Publishing, 1997.

[99] A. S. Nastase, "How to Derive the RMS Value of Pulse and Square Waveforms," *MasteringElectronicsDesign.com*, 2012. [Online]. Available: https://masteringelectronicsdesign.com/how-to-derive-the-rms-value-of-pulse-and-square-waveforms/ (Accessed 5/12/2018).

[100] K. C. Pohlmann, *Principles of Digital Audio*, 6th ed., pp. 20—23. New York City: McGraw-Hill, 2011.

[101] Image-Line, "Edison," *FL Studio 20 Reference Manual*. [Online]. Available: https://www.image-line.com/support/flstudio_online_manual/html/plugins/Edison.htm (Accessed 5/12/2018).

[102] "Why not always cut the 20-30 Hz range?" *KVR Audio*, 2009–2011. [Online]. Available: https://www.kvraudio.com/forum/viewtopic.php?f=62&t=313807&sid=7da7241355268614d8690cad3702890b (Accessed 9/12/2018).

[103] S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, pp. 261–276. San Diego, CA, USA: California Technical Publishing, 1997.

[104] S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, pp. 351–353. San Diego, CA, USA: California Technical Publishing, 1997.

[105] E. C. Everbach, "Noise Quantification and Monitoring: An Overview," *The Science Building Project*, 2000. [Online]. Available: http://www.swarthmore.edu/NatSci/sciproject/noise/noisequant.html (Accessed 25/11/2018).

[106] utz, "Tutorial: How to Write a 1-Bit Music Routine," *1-Bit Forum*, July 2015. [Online]. Available: http://randomflux.info/1bit/viewtopic.php?id=21 (Accessed 04/04/2017).

[107] blargg, "NES APU Sound Hardware Reference," nesdev.com, 2004. [Online]. Available: http://nesdev.com/apu_ref.txt (Accessed 29/03/2019).

[108] "Commodore MOS Technology IMMOS." [Online]. Available: http://archive.6502.org/datasheets/mos_6581_sid.pdf (Accessed 29/03/2019).

[109] C. R. Nave, "Clarinet Waveform," 1998. [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/Music/clarw.html (Accessed 14/12/2018).

[110] B. Bland, "Making Complex Waves," 1999. [Online]. Available: http://hep.physics.indiana.edu/~rickv/Making_complex_waves.html (Accessed 16/12/2018).

[111] T. Rutherford-Johnson, Ed., p. 268. Oxford University Press, 2013.

[112] D. Strange, "ADSR Envelope Generator," vol. 2, February 1984, (Accessed 19/02/2017).

[113] Software Creations, "The Sentinel," ZX Spectrum, 1987.

[114] M. Ltd, "Chronos," ZX Spectrum, 1987.

[115] Proxima Software, "Orfeus Music Assembler," ZX Spectrum, 1990.

[116] J. Deak, "ZX-3," *World of Spectrum*, 1991. [Online]. Available: http://www.worldofspectrum.org/infoseekid.cgi?id=0027576 (Accessed 29/03/2019).

[117] P. Ball, *The Music Instinct*, p. 69. London, UK: The Bodley Head, 2010.

[118] D. Levitin, *This is Your Brain on Music: The Science of a Human Obsession*, pp. 69—70. Dutton, 2006.

[119] D. Levitin, *This is Your Brain on Music: The Science of a Human Obsession*, pp. 53—54. Dutton, 2006.

[120] P. Clarke, "Ocean Loader 3," Commodore 64 Loading Software, 1987.

[121] B. Marshall, "Last Ninja 2," ZX Spectrum, 1988.

[122] Raphaelgoulart, "Chromospheric Flares," *ZXART*, 2014. [Online]. Available: https://zxart.ee/eng/authors/m/mister-beep/chromospheric-flares/

[123] D. A. Russell, "Acoustics and Vibration Animations," *The Pennsylvania State University*, July 1996. [Online]. Available: https://www.acs.psu.edu/drussell/demos/superposition/superposition.html (Accessed 9/12/2018).

[124] H. E. Haber, "How to add sine functions of different amplitude and phase," 2009. [Online]. Available: http://scipp.ucsc.edu/~haber/ph5B/addsine.pdf (Accessed 9/12/2018).

[125] J. Corey, *Audio Production and Critical Listening: Technical Ear Training*, ser. Audio Engineering Society Presents, pp. 104—105. Taylor & Francis, 2016.

[126] G. Davis, G. Davis, R. Jones, and Y. Corporation, *The Sound Reinforcement Handbook*, ser. Recording and Audio Technology Series, pp. 81—86. Hal Leonard, 1989.

[127] "MSD 101: Pulse-Width Modulation," *Motion System Design*, October 2000.

[128] Utz, "utz82/ZX-Spectrum-1-Bit-Routines," *GitHub*, November 2018. [Online]. Available: https://github.com/utz82/ZX-Spectrum-1-Bit-Routines/tree/master/stringks (Accessed 22/12/2018).

[129] D. T. Carter, "Rockman," ZX Spectrum, 1987.

[130] L. Games, "The Secret Of Monkey Island," DOS Computer Game, 1990.

[131] Utz, "Sound Routines," *irrlicht project - code*. [Online]. Available: http://irrlichtproject.de/code.php (Accessed 22/12/2018).

[132] Raphaelgoulart, "surprisingly NOT four twenty," *ZXART*, 2014. [Online]. Available: https://zxart.ee/eng/authors/r/raphaelgoulart/surprisingly-not-four-twenty/ (Accessed 29/03/2019).

[133] Brink, "M'Lady," *ZXART*, 2014. [Online]. Available: https://zxart.ee/eng/authors/b/johan-elebrink/mlady/ (Accessed 29/03/2019).

[134] P. B. Todd and S. A. Lakawicz, "Interview with David Warhol (composer, programmer)," *Video Game History*, December 2016. [Online]. Available: http://www.vgarc.org/vgarc-originals/interview-with-david-warhol/ (Accessed 24/04/2017).

[135] T. Follin, "Star Tip 2," *Your Sinclair*, vol. 20, pp. 201–213, August 1987.

[136] "Interview with David Wise (December 2010)," *Square Enix Music Online*, 2010. [Online]. Available: https://www.squareenixmusic.com/features/interviews/davidwise.shtml (Accessed 30/03/2019).

[137] P. Phelps, "A modern implementation of chiptune synthesis," University of the West of England, 2007. [Online]. Available: https://woolyss.com/chipmusic/chipmusic-discovery/PhillPhelps-ChiptuneSynth.pdf (Accessed 04/04/2017).

[138] N. Baldwin, "NES Audio Tools," 2011. [Online]. Available: http://nes-audio.com/ (Accessed 24/04/2017).

[139] J. Cauldwell, "Loudspeaker Sound Effects," *How To Write ZX Spectrum Games –Chapter 3*, Bytes: Chuntey, February 2013. [Online]. Available: https://chuntey.wordpress.com/2013/02/28/how-to-write-zx-spectrum-games-chapter-3/ (Accessed 04/04/2017).

[140] J. Deak, "ZX-3," ZX Spectrum, 1991.

[141] C. Roads, *Microsound*, paperback ed., pp. 1–43. Cambridge, Massachusetts: MIT Press, 2004.

[142] T. Wishart, *Audible Design*, 6th ed. York: Orpheus the Pantomime, 1994.

[143] K. C. Pohlmann, *Principles of Digital Audio*, 6th ed., pp. 693–729. New York City: McGraw-Hill, 2011.

[144] D. Dodson, "Composing for 1-bit Microchip: Tristan Perich," *The 1-Bit Forum*, August 2010. [Online]. Available: http://cdm.link/2010/08/composing-for-1-bit-microchip-tristan-perich/ (Accessed 07/04/2017).

[145] STMicroelectronics, "General-purpose timer cookbook," *AN4776 Application note*, June 2016. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/application_note/group0/91/01/84/3f/7c/67/41/3f/DM00236305/files/DM00236305.pdf/jcr:content/translations/en.DM00236305.pdf (Accessed 07/04/2017).

[146] Shiru, "Tritone on Arduino," *The 1-Bit Forum*, February 2017. [Online]. Available: http://randomflux.info/1bit/viewtopic.php?id=126 (Accessed 07/04/2017).

[147] Atmel, "Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers," *8-bit Atmel Microcontrollers: Application Note*, p. 6. [Online]. Available: http://www.atmel.com/images/doc8453.pdf (Accessed 01/04/2017).

[148] O. Online, "Microcomputer, n." Oxford University Press, June 2018. [Online]. Available: www.oed.com/view/Entry/117935 (Accessed 20/06/2018).

[149] O h ! 石, "MZ-80K," RetroPC.net, 2009. [Online]. Available: http://www.retropc.net/ohishi/museum/mz80k.htm (Accessed 29/04/2018).

[150] T. Matsushima, *Beyond MIDI: The Handbook of Musical Codes*, E. Selfridge-Field, Ed., pp. 143–145. Cambridge, MA, USA: MIT Press, 1997.

[151] *Computing Japan*. LINC Japan, 1999, no. v. 54-59. [Online]. Available: https://books.google.co.uk/books?id=oP61AAAAIAAJ

[152] Karl-Heinz, "The monitor program SP-1002," www.sharpmz.org, September 2002. [Online]. Available: https://www.sharpmz.org/mz-80k/monisubprg.htm (Accessed 30/04/2018).

[153] C. Walshaw, “The ABC Music Standard 2.1,” abcnotation.com, July 2015. [Online]. Available: http://abcnotation.com/wiki/abc:standard:v2.1 (Accessed 18/05/2018).

[154] C. Walshaw, “A Brief History Of ABC,” abcnotation.com, 2017. [Online]. Available: http://abcnotation.com/history (Accessed 18/05/2018).

[155] Manbow-J and J. ’Virt’ Kaufman, “MCKC: MML > MCK Converter Ver 0.14,” 2002. [Online]. Available: http://www.geocities.co.jp/Playtown-Denei/9628/mck/mckc-e.txt (Accessed 29/04/2018).

[156] Woolyss, “MML,” 2016. [Online]. Available: https://woolyss.com/chipmusic-mml.php#mml (Accessed 29/04/2018).

[157] ALOE, “マビノギ MML 作曲ツール 3ML EDITOR 2 Webpage,” 2008. [Online]. Available: http://3ml.jp/ (Accessed 29/04/2018).

[158] D. Farler, “Ultimate PPMCK MML Reference,” August 2007. [Online]. Available: http://www.shauninman.com/assets/downloads/ppmck_guide.html (Accessed 29/04/2018).

[159] Ｏｈ！石, 菅, and (hally) et al., “シャープ博物館,” RetroPC.net, 2009. [Online]. Available: http://www.retropc.net/ohishi/museum/ (Accessed 30/04/2018).

[160] J. O’Doherty, “MMLShare,” MMLShare, 2018. [Online]. Available: https://www.mmlshare.com/ (Accessed 30/04/2018).

[161] “Getting started with MML (mck, ppmck),” nesdev.com, 2016. [Online]. Available: https://forums.nesdev.com/viewtopic.php?f=6&t=14774 (Accessed 30/04/2018).

[162] “GETTING STARTED WITH MML (MCK,PPMCK),” Chipmusic.org, 2016. [Online]. Available: https://chipmusic.org/forums/topic/18991/getting-started-with-mml-mckppmck/ (Accessed 30/04/2018).

[163] micol972, “xpmck,” June 2011. [Online]. Available: http://jiggawatt.org/muzak/xpmck/ (Accessed 29/04/2018).

[164] “MML,” Mabinogi World Wiki, August 2017. [Online]. Available: https://wiki.mabinogiworld.com/view/MML (Accessed 30/04/2018).

[165] J. Kotlinski, “Little Sound DJ,” Game Boy Software. [Online]. Available: http://www.littlesounddj.com (Accessed 01/04/2019).

[166] Thunder, “MODFIL10.TXT,” *File Formats Reverse Engineering*. [Online]. Available: http://lclevy.free.fr/mo3/mod.txt (Accessed 01/04/2019).

[167] jsr, “Famitracker,” Sequencer Software, 2005—2015. [Online]. Available: http://famitracker.com/# (Accessed 01/04/2019).

[168] M. W. Butterfield, “The Power of Anacrusis: Engendered Feeling in Groove-Based Musics,” *Music Theory Online*, vol. 12, no. 4, 2006.

[169] E. Selfridge-Field, Ed., *Beyond MIDI: The Handbook of Musical Codes*. Cambridge, MA, USA: MIT Press, 1997.

[170] H. Hinrichsen, "Revising the Musical Equal Temperament," Universit¨at W¨urzburg, Fakult¨at f¨ur Physik und Astronomie, November 2015. [Online]. Available: https://arxiv.org/pdf/1508.02292.pdf (Accessed 29/09/2018).

[171] micol972, "XPMCK - Cross Platform Music Compiler Kit," *XPMCK manual*, 2011. [Online]. Available: http://jiggawatt.org/muzak/xpmck/manual.html

[172] J. P. Crutchfield, "The Calculi of Emergence: Computation, Dynamics, and Induction," *Physica D*, vol. 75, no. 6, pp. 11––54, 1994. [Online]. Available: http://jn.physiology.org/content/83/6/3548

[173] R. Lopez-Ruiz, H. Mancini, and X. Calbet. (2010, September) A Statistical Measure of Complexity. [Online]. Available: https://arxiv.org/pdf/1009.1498.pdf (Accessed 17/08/2019).

[174] P. Ball, *The Music Instinct*, p. 119. London, UK: The Bodley Head, 2010.

[175] S. Rickard, "The Beautiful Math Behind The World's Ugliest Music," TEDxMIA, 2011. [Online]. Available: https://www.ted.com/talks/scott_rickard_the_beautiful_math_behind_the_ugliest_music (Accessed 30/09/2018).

[176] J. Davies, *The Psychology of Music*. Stanford University Press, 1978. [Online]. Available: https://books.google.co.uk/books?id=4cFBV9cykHsC

[177] B. Oliver, "Mr. turquoise synth," March 2017. [Online]. Available: https://eprints.soton.ac.uk/406534/

[178] N. C. Department, "The Legend of Zelda: A Link to the Past," Super Nintendo Entertainment System Software, 1991.

[179] D. Levitin, *This is Your Brain on Music: The Science of a Human Obsession*, p. 155. Dutton, 2006.

[180] M. Ltd, "Agent X," ZX Spectrum, 1987.

[181] M. Ltd, "Raw Recruit," ZX Spectrum, 1987.

[182] I. Software, "Vectron," ZX Spectrum, 1985.

[183] D. A. Jaffe, "Orchestrating the Chimera: Musical Hybrids, Technology and the Development of a "Maximalist" Musical Style," *Leonardo Music Journal*, vol. 5, pp. 11–18, 1995. [Online]. Available: http://www.jstor.org/stable/1513155

[184] R. Kurth, "An Introduction to the Music of Milton Babbitt," *Intégral*, vol. 8, pp. 147–182, 1994. [Online]. Available: http://www.jstor.org/stable/40213958

[185] M. Connor, "The Impossible Music of Black MIDI," *Rhizome*, September 2013. [Online]. Available: http://rhizome.org/editorial/2013/sep/23/impossible-music-black-midi/ (Accessed 01/04/2019).

[186] J. Hocker, "List of Works," *Der Komponist Conlon Nancarrow*. [Online]. Available: http://www.nancarrow.de/list_of_works_english_version.htm

[187] Atmel, "tinyAVR Microcontrollers," *atmel.com*. [Online]. Available: http://www.atmel.com/products/microcontrollers/avr/tinyavr.aspx (Accessed 04/05/2017).

[188] Konami, "Skate Or Die II," Nintendo Entertainment System Software, 1988.

[189] Novotrade, "Blades Of Steel," Nintendo Entertainment System Software, 1988.

[190] R. Allain, "Why Are Songs on the Radio About the Same Length?" *Wired*, June 2017. [Online]. Available: https://www.wired.com/2014/07/why-are-songs-on-the-radio-about-the-same-length/ (Accessed 19/05/2019).

[191] K. McKinney, "A hit song is 3 to 5 minutes long. Here's why." *Vox*, January 2015. [Online]. Available: https://www.vox.com/2014/8/18/6003271/why-are-songs-3-minutes-long (Accessed 19/05/2019).

[192] "KVR Forum: Length of "Average" Full-length Music CD/Album(LP)," *KVR Audio*, September 2006. [Online]. Available: https://www.kvraudio.com/forum/viewtopic.php?t=151744 (Accessed 19/05/2019).

[193] Lieff, "lieff/minimp3," *GitHub*, March 2019. [Online]. Available: https://github.com/lieff/minimp3

[194] J. Powell, *Why You Love Music: From Mozart to Metallica–The Emotional Power of Beautiful Sounds*, pp. 64—67. Little, Brown, 2016.

[195] C. Seeger, "On the Moods of a Music-Logic," *Journal of the American Musicological Society*, vol. 13, no. 1/3, pp. 224–261, 1960. [Online]. Available: http://www.jstor.org/stable/830257

[196] M. Fritsch, *"History of Video Game Music"*, pp. 11—41, 08 2012.

[197] J. Powell, *Why You Love Music: From Mozart to Metallica–The Emotional Power of Beautiful Sounds*, pp. 54—64. Little, Brown, 2016.

[198] E. Margulis, *On Repeat: How Music Plays the Mind*. OUP USA, 2014.

[199] N. C. Department, "Super Mario Bros." Nintendo Entertainment System Software, 1985.

[200] N. Falcom, "Legacy Of The Wizard," Nintendo Entertainment System Software, 1989.

[201] H. S. . Creatures, "Pokémon Trading Card Game," Game Boy Software, 1998.

[202] "ZXART Music," *ZXART*, 2019. [Online]. Available: https://zxart.ee/eng/music/ (Accessed 2019).

[203] M. Smith, "Manic Miner," ZX Spectrum, 1983.

[204] Nintendo R&D1, "Donkey Kong," Arcade Cabinet, 1981.

[205] J. Kornmeier and M. Bach, "The Necker cube—an ambiguous figure disambiguated in early visual processing," vol. 8, no. 45, pp. 955—960, 2005, (Accessed 19/02/2017).

[206] Capcom, "Ducktales," Nintendo Entertainment System Software, 1989.

[207] M. Changizi, "Harnessing vision for computation," *Perception*, vol. 37, pp. 1131–1134, 2008.

[208] T. Rutherford-Johnson, Ed., p. 141. Oxford University Press, 2013.

[209] B. Newbould. (2001) Palindrome. [Online]. Available: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000041238 (Accessed 19/09/2019).

[210] B. Newbould. (2001) Mirror forms. [Online]. Available: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000041239 (Accessed 19/09/2019).

[211] S. Kostka, *Materials and Techniques of Twentieth-century Music*, p. 316. Pearson Prentice Hall, 2006. [Online]. Available: https://books.google.co.uk/books?id=5YQJAQAAMAAJ

[212] J. Colannino, F. Gómez, and G. Toussaint, "Analysis of emergent beat-class sets in steve reich's clapping music and the yoruba bell timeline," p. 2, 02 2019.

[213] P. Flajolet and M. Soria, "The cycle construction," *SIAM J. Discrete Math.*, vol. 4, pp. 58–60, 1991.

[214] B. Eno, "Generative Music," In Motion Magazine, July 1996. [Online]. Available: http://www.inmotionmagazine.com/eno1.html (Accessed 19/02/2017).

[215] J. Goldstein, "Emergence as a Construct: History and Issues," *Emergence*, vol. 1, no. 1, pp. 49–72, 1999.

[216] Josephmisiti, "Awesome Machine Learning," *GitHub*, January 2019. [Online]. Available: https://github.com/josephmisiti/awesome-machine-learning (Accessed 19/1/2019).

[217] I. Xenakis, *Formalized Music: Thought and Mathematics in Composition*, ser. Harmonologia series. Pendragon Press, 1992. [Online]. Available: https://books.google.co.uk/books?id=y6lL3I0vmMwC

[218] V.-M. Heikkilä, "Experimental music from very short C programs," *YouTube*, September 2011. [Online]. Available: https://www.youtube.com/watch?v=GtQdIYUtAHg&t=137s (Accessed 19/1/2019).

[219] Kragen, "Bytebeat," *The Canonical Hackers*. [Online]. Available: http://canonical.org/~kragen/bytebeat/ (Accessed 19/1/2019).

[220] V. Heikkilä, "Some Deep Analysis Of One Line Music," *Some deep analysis of one-line music programs.* [Online]. Available: http://countercomplex.blogspot.com/2011/10/some-deep-analysis-of-one-line-music.html

[221] V. Heikkilä, "Discovering novel computer music techniques by exploring the space of short computer programs," *CoRR*, vol. abs/1112.1368, 2011. [Online]. Available: http://arxiv.org/abs/1112.1368

[222] B. Carnahan and J. Wilkes, *Digital Computing, FORTRAN IV, WATFIV, and MTS (with *FTN and *WATFIV) /by Brice Carnahan, James O. Wilkes*, ser. Digital

Computing, FORTRAN IV, WATFIV, and MTS (with *FTN and *WATFIV) /by Brice Carnahan, James O. Wilkes. Chemical Engineering Department, University of Michigan, 1978, no. v.1. [Online]. Available: https://books.google.co.uk/books?id=tvnuAAAAMAAJ

[223] B. W. Kernighan and D. M. Ritchie. PRENTICE HALL, Englewood Cliffs, New Jersey 07632, 1988.

[224] M. Stamp. (2003, July) Once Upon a Time-Memory Tradeoff. [Online]. Available: http://www.cs.sjsu.edu/faculty/stamp/RUA/TMTO.pdf (Accessed 17/08/2019).

[225] Google. (2016, May) SyntaxNet. [Online]. Available: https://ai.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html (Accessed 10/09/2019).

[226] J. L. Borges, "The Library of Babel," in *in Borges J.L. Collected Fictions (Penguin)*, 1999.