

- UFEE63-30-3 -

A modern implementation of chiptune synthesis

by

Philip Phelps (#03974520)

Summary

A chiptune can be broadly categorised as a piece of music produced by sound chips in home computer/ games systems popular in the 1980s into the 1990s. The synthesis techniques often employed in chiptunes involve the careful control of the hardware features available in the sound chips. Such chips generally featured a limited number of simple oscillators with simple waveforms such as square, triangle, pseudorandom noise, and so on. In order to create more interesting sounds, chiptune composers rely on software synthesis structures that are used to configure the sound chip to alternate between waveforms, and to alter the pitch of the oscillators.

MaxMSP is a graphical programming environment developed by Cycling'74. This report documents the reasons behind the decision to choose MaxMSP for the design and implementation of a software synthesiser for the creation of chiptunes.

The report presents the detailed research into the sound hardware, and software used for composing chiptunes, exposing the differences in user interface design between modern composition software tools such as MIDI sequencers and historic composition tools such as trackers. Several pieces of historic software are examined; with elements of each user interface being assessed for inclusion in a modern software synthesiser user interface. The report also explains in detail the internal hardware structure of several popular sound chips (such as the General Instruments AY-3-8910) and examines several examples of software structures used to control the hardware synthesis structures inside the chips.

The report documents the design of numerous software modules (programmed in the MaxMSP environment) and explains how each module can be combined to form the software synthesiser. Several block diagrams are included, explaining the internal structure of various modules, along with detailed maintenance notes for each module to help other programmers continue development of the system.

The report assesses the computational efficiency of the produced synthesiser, and suggests several possible improvements that could be made in this area. The development of an “external” MaxMSP object in the C programming language is also documented, with fully commented source code provided.

Acknowledgements

The author would like to thank Dr. David Creasey for his support and helpful comments during the project, and Dr. Barbara Savage for the conversations that inspired several diagrams in the introduction chapter.

The author would also like to thank Gareth Morris (gwEm) for the guidance regarding Atari chiptune terminology, Max Matthews, Miller Puckette and the Cycling'74 team for creating MaxMSP, and to my mother for help during the proof reading.

Table of contents

Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Chiptune sound character	5
1.3 Introduction to MaxMSP and PureData	5
1.4 Objectives	7
1.5 Strategy	8
1.5.1 Design strategy	8
1.5.2 Project management strategy	11
1.5.3 Software development strategy	13
1.5.4 Report structure management	15
1.5.5 Testing strategy	16
1.6 A brief overview of MIDI	18
1.7 Trackers vs. MIDI sequencers	19
1.8 Layered systems	22
Chapter 2 Interface considerations	25
2.1 Choosing the development environment	25
2.1.1 Environment decision	27
2.2 Historic interfaces	27
2.2.1 Historical sound design interface influences	28
2.3 Modern interfaces	34
2.3.1 Genetic algorithm	34
2.3.2 Possible solution to “tracker instrument numbers” problem	35
2.3.3 Parameter storage	36
Chapter 3 Synthesis considerations	37
3.1 Historic hardware synthesis	41
3.2 Historic software synthesis	42
3.2.1 Software with similar goals to the project	45
3.3 Modern techniques	46
Chapter 4 Summary of findings	47
4.1 Major decisions - changes to objectives	47
4.2 Summary of findings	48
4.3 Specifications and requirements	49
Chapter 5 Implementation	49
5.1 MaxMSP terms and conventions	51
5.1.1 Fundamental concepts and terms	51
5.1.2 Commonly used objects	53
5.2 Overview of system	55
5.3 Synthesis	57
5.3.1 The [clockedseq~] abstraction	60
5.3.2 Arpeggiation and Portamento [arpporta~] abstraction	67
5.3.3 The [midiarpeggiator~] abstraction	70
5.3.4 The [midiportamento~] abstraction	73
5.3.5 The [lfo~] abstraction	75
5.3.6 Oscillators	77
5.3.7 Amplitude envelope structure	80

5.3.8 Ringmodulation.....	82
5.3.9 The [envelope~] abstraction	85
5.4 Interface components	88
5.4.1 The [Interface] abstraction.....	88
5.4.2 [midinotefilter] abstraction	91
5.4.3 [filterprocessing] abstraction	92
5.5 [editor] GUI abstraction.....	95
5.5.1 Control methods.....	96
5.5.2 Step sequencers	97
5.5.3 Envelopes.....	100
5.5.4 [calculator] patch.....	101
5.5.5 [mixer~] patch.....	102
5.6 Communication and storage of parameters	105
5.6.1 [transportdata] abstraction.....	108
5.6.2 [convertdata] abstraction.....	109
5.7 Improving computational efficiency.....	110
5.8 Custom MaxMSP external.....	112
5.8.1 Requirements specification.....	112
5.8.2 Implementation	113
5.8.3 Problems and abandonment.....	115
Chapter 6 Conclusions	115
6.1 Content summary	117
6.2 Time planning.....	118
6.3 Further work	119
6.4 Fulfilment of objectives.....	121
6.4.1 List of completed objectives	121
6.4.2 List of uncompleted objectives	121
6.5 List of achievements.....	122
6.5.1 Project achievements	122
6.5.2 Learning achievements.....	122
Appendices	
Appendix 1 - Completed project proposal form.....	125
Appendix 2 - Completed progress report form	126
Appendix 3 - Completed progress monitoring form	127
Appendix 4 - Sound chip hardware research	128
Appendix 5 - Editor abstraction	137
Appendix 6 - External MaxMSP object [clockedindex~] development	138
Appendix 8 - Time planning diagrams, Gantt charts	142

Chapter 1 Introduction

This chapter explains background information along with the various reasons and motivation for choosing the project. The project objectives are also outlined here. Explanations of the various strategies taken during design, development, testing, and report writing are given. The MIDI system is discussed briefly, along with an examination of a class of computer based composition software known as a “tracker”.

When writing the report, the author found that a great deal of information was referred to time and time again. To reduce the need for repetition in the text, many of the core concepts are introduced in this chapter, these are expanded in much more detail later.

1.1 Motivation

In the 1980s into the 1990s, custom sound chips were used mainly in home computer systems and arcade machines. In order to relieve the load placed on CPUs (Central Processing Units) in these systems, sound chips were used as co-processors in much the same way as a GPU (Graphics Processing Unit) is used today for high performance 3D graphics. The sound chips were often very basic “state machines” that could be configured via the CPU to alter parameters of a limited number of internal oscillators such as the waveform, pitch, and so on. Music composed for these systems during the 1980s and 1990s is nostalgically referred to as “chiptune” music, derived from the fact that hardware chips are being used to produce the music.

Towards the end of these chips popularity in computer systems, certain software and sound design techniques had become standardised. Normally a tracker is the most popular type of composition software for these types of systems.

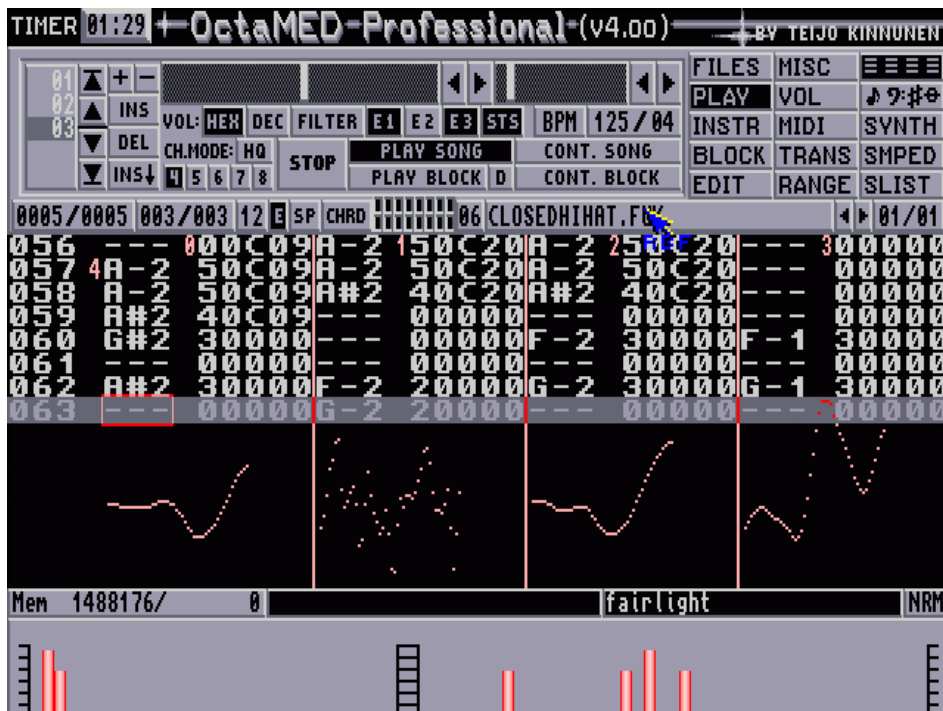


Figure 1.1 - An example of a tracker (OctaMED v4.0)

In a tracker, step sequences for note triggers are arranged on vertical scrolling *tracks* (like a player-piano) that correspond to internal hardware *channels*. Tracker software is generally not a live performance tool. With a tracker, music is programmed in step-time line-by-line, note-by-note. Synthesis parameters such as the active waveform generator are controlled by separate *step sequences* or *control tables*, which most trackers refer to as *instrument definitions* since they determine how a given note trigger will sound.

During the time when games were being written for these computer systems; often the composers writing music for the games were also experienced programmers. New disciplines of innovative sound design and music production techniques were developed to work around hardware limitations. The programmers responsible for the software that enabled sound programming on these home computers tried to encompass every feature of the sound chips they control, aiming to allow the user to program music down to every last detail.

Tracker software is full of programming conventions applied to music:

- Individual notes are equivalent to programming statements
- Musical phrases are equivalent to blocks of code

There are more explicit programming analogies:

- There are structures to allow branching to other blocks
 - Equivalent in programming to “GOTO” or “JMP” statements
- There are structures to allow repetition of blocks without manually duplicating
 - Equivalent in programming to a “FOR UNTIL” structure or a musical “coda”
- Some trackers allow modulation of certain musical parameters such as amplitude and transposition from a root pitch via separate sequences
 - Equivalent in programming to “nesting” routines

Trackers are “hyper sequencers”; a single note programmed in a tracker can trigger a sub-sequence of oscillator waveforms, whilst also triggering sequences of modulation parameters (transposition, pulse width, arpeggiations based on the root note, and so on).

It would be an understatement to say that the programming of these parameters individually (by writing the instructions to control the sound chip for each note in a large piece of music) would be tedious. In some cases, programmer/composers of game music did indeed practice their craft this way, constructing their own “replay routines” in assembler to control the chips. Tracker software aims to relieve the user from the burden of having to configure the sound chip registers directly for each note trigger, allowing the user to concentrate on the musical aspects of choosing the instrumentation, and composing the music.

The author was exposed to the sounds associated with chiptunes at an early age: by playing computer games. Later in life, an interest in music composition, electronic music, and sound design using computers led to the author’s discovery of software composition tools (the trackers described earlier) that could be run on a home computer. As time progressed, and the author migrated from one computer system to another, the composition tools changed, eventually moving into MIDI sequencers.

The main motivation for this project is that modern audio sequencing software and synthesizers place a different emphasis on the way sounds are produced. There are very few programmable synthesizers commonly available that allow such low level configuration as was

possible when using a computer system of the past featuring one of these sound chips. In order to reproduce the characteristic chiptune timbres that swap waveforms and modulate parameters with such exacting precision, the user must spend time programming the exact instructions for each note; a job that in the past, trackers were able to do with ease.

As attractive as the “hyper sequencing” capabilities of trackers are, modern sequencers have certain desirable characteristics too. The main benefit, in the author’s opinion, is a greater flexibility in terms of visibility of components involved in the composition. Most MIDI sequencers allow the user to present the composition graphically as blocks on a timeline where each block represents a musical phrase or section. Composition decisions that are concerned with how to re-structure a complicated musical piece are made easily, since the blocks are easily identifiable. For users who do not read music notation; a graphical representation of a piano keyboard with note triggers represented as blocks running horizontally in time is often available. The contour of a melody is instantly recognisable as the vertical pitches and horizontal durations of notes. Re-harmonising a chord is a simple matter of shuffling the vertical blocks on screen as in the figure below.

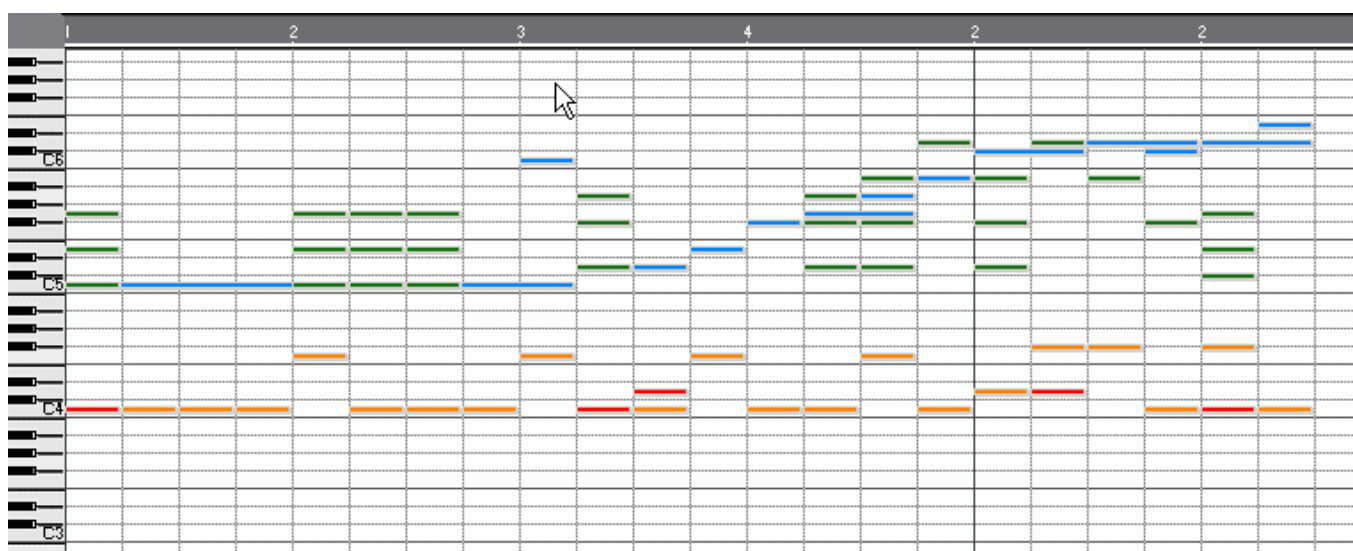


Figure 1.2 - An example timeline display from a modern sequencer (Cubase v5.0)

In contrast, the limitations of the computer systems of the past meant that displays were mostly textual. Tracker software invariably presents note pitches as strings of alphanumeric codes “A#3” for “note A sharp, in octave 3”, and cryptic hexadecimal codes representing the commands to bend notes, or add vibrato. Limited screen resolutions prevented the richly visual representation of data common in modern graphical user interfaces.

It was previously mentioned that the general approach in trackers is to refer to the synthesis parameters for a certain sound as an *instrument definition*. Each instrument is assigned a number for easy reference during composition. When a composer begins entering note triggers into the tracker, each note trigger is associated with an instrument number. During playback of notes, the tracker software loads the appropriate synthesis parameters for the required instrument into the hardware registers.

Imagine this analogy of a tracker designed to control a four-channel sound chip: A quartet of musicians are standing on stage, they are about to play a piece of music. All four musicians

have been given an identical set of monophonic instruments. The players are reading from a score that dictates which note to play on which instrument. When they start playing, they can only pick up one instrument at a time, and since all instruments are monophonic, only four notes can sound from the stage at any one time.

At first, this limitation of four-note polyphony might seem to be a big problem. It appears at first glance that a piece of music composed with this arrangement could perhaps assign each player an instrument at the beginning of the piece and from then on only dictate which notes to play.

However, the reader will appreciate that this is an analogy to a computer controlled situation, and it is possible to compose an extremely complicated piece where each player swaps between the available instruments the instant before they play a note, and continues to swap instruments before each and every note. This would create the impression of a much larger band playing a much wider variety of instruments. The four-note polyphony limitation is reduced in importance when music is composed in this way.

Composing a score with hundreds of instrument swaps could be very complicated for a human to comprehend. A sheet of music notation could easily be filled with hundreds of written instructions written next to each note. The tracker approach to ease composition of music that swaps instruments between notes is to display each note trigger with an accompanying instrument indicator. This enables a human composer to see the important information at a glance. See Table 1.1 below:

Player 1 instructions	Player 2 instructions
Kick drum	Violin, octave 2, note E
(Rest)	Violin, octave 4, note G
Trumpet, octave 3, note C	Snare drum
(Rest)	High hat pedal closed
Piano, octave 3, note C	Violin, octave 3, note G
Piano, octave 5, note F	High hat pedal open

Table 1.1 - Tracker instructions

Viewing these instructions it is easy to see which note triggers occur simultaneously and on what instrument they are to be played.

It is very important to make clear that modern MIDI sequencers do not display this sort of information very well.

A user composing chiptunes on a modern system using the project is likely to be using a MIDI sequencer and could very easily be confused if the MIDI sequencer does not allow the user to display program change information next to the note trigger.

The problem essentially boils down to the need for providing an elegant method of controlling low-level synthesis parameters during composition (and playback) that will not be confusing to the user. Designing a solution to this problem will have an impact on the synthesis structure, the machine interface (MIDI sequencing) and on the user interface. The solutions related to each area are discussed in the appropriate chapter of the report.

Another problem is expression, that is to say communication of emotion by rendering a musical phrase in a specific way such as slurring or bending notes, vibrato or tremolo. The term “expression” here implies some sort of modulation or deviation from the core pitches and rhythm of the composed notes.

Trackers often display note bends and other “expression” qualities such as vibrato, tremolo, arpeggiations, etc, next to note triggers in a similar way to instrument identifiers, for easy identification and adjustments during composition, whereas MIDI sequencers often avoid placing such information next to notes in the same manner as trackers.

It is interesting to note that this problem of expression has bothered synthesised music composers in general, that is to say chiptune composers are not the only people affected by this. Synthesiser interface designers have long struggled to create machines with comparable degrees of the expression possible on acoustic instruments. The reader will appreciate that a synthesiser is a very configurable device, and that by its very nature every aspect of the sound it makes is specified exactly. Logically then, it ought to be more expressive than any acoustic instrument. Various human and machine interface technologies are available, but synthesisers in general remain difficult to control expressively.

The overall intention of the project is to combine elements from historic systems with elements of modern systems; enabling the creation of music with the programmed sound character of historic chiptune timbres using modern composition tools. However, as explored in more detail elsewhere, there may be problems as a result of attempting to combine old sound design methods with new composition systems.

1.2 Chiptune sound character

Some characteristic qualities of chiptunes are:

- Low bit-depth waveforms “mathematical” waveforms, often 8 or even 4-bit, squarewave, ramps (saw, triangle), pseudorandom noise.
- Fast switching between waveforms in an effort to create more complex waveforms, adding interest to sounds.
- Limited polyphony often subverted by playing extremely fast arpeggios in place of using simultaneous oscillators for chords.

Although this is an extremely broad overview, it is included here to help the reader become familiar with the desired sound character. The report expands upon each of these areas in much more depth later.

1.3 Introduction to MaxMSP and PureData

The original idea for the project began as an experimental prototype, developed during the summer of 2006, much before the final year of study began in September. This prototype was developed in a programming environment called PureData.

Later, in the early stages of the project, there were two particular pieces of software that the author had in mind for further project development; PureData and MaxMSP. PureData was the natural choice since the prototype was developed in this environment, but both could be described as a “rapid-prototyping modular graphical programming environments”. They both use graphical representations, manipulated with the mouse, to represent the

algorithms to be carried out by the computer rather than textual representations, typed with a keyboard.

Max/MSP is a “Visual programming environment for music, audio, and new media” (Cycling’74, 2006). Max/MSP and PureData are very similar in nature, due to the involvement of Miller Puckette in both projects. Max/MSP is a “closed source” commercial product, whilst PureData is an “open source” project and is free to use.

Both systems (Max/MSP and PureData) operate in a similar modular way, with blocks called “objects” representing processing modules and connecting cables representing conduits that allow the blocks to exchange information. An analogy can be drawn with a modular analogue synthesiser, with modules connected together to form a greater whole. The user interface of both systems is designed in such a way that the graphical representation of blocks and cables represent the flow of data. PureData’s website proudly displays the phrase “the diagram is the program” (Puredata.org, 2006).

A distinction between “control rate” and “audio rate” signals is also made in both systems, this is to simplify the scheduling of computational threads that must output values at “audio rates” (i.e. tens of thousands of times per second) versus “control rate” signals that can afford to be computed at a much lower rate.

Both systems are legitimate and complex programming languages, special emphasis should be placed on the word “programming” here, their visual and graphical nature does not make them particularly easy; In fact, in some cases, their graphical nature can make programming certain structures more difficult. At first glance, their graphical nature may imply that the programmer is at an advantage with “things being done for him/her”. The author should stress that this is definitely not the case.

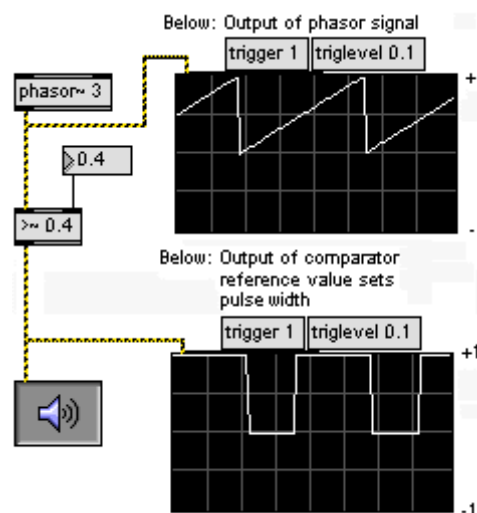


Figure 1.3 - Example MaxMSP patch

The MaxMSP/PureData programmer is presented with a blank canvas and just as with a traditional textual programming language, the complex structure of the programmer’s creation must be built up from more primitive structures. Both Max/MSP and PureData allow the programmer to specify things at an extremely low level down to moving data between registers, delaying samples in time domain buffers, modification of values held in registers by

addition, multiplication, and masking binary values. The collection of objects that perform these low level functions are complemented by a set of objects that perform high level functions with a particular music synthesis slant such as high order band-pass filters, envelope generators, fast Fourier transforms and frequency shifting.

In both systems, there is a set of “standard” objects called “primitives”. Their name is not intended to be derogatory; the word simply separates them from custom objects called “externals”. Both systems include an API that enables almost limitless extension by writing new “external” objects in the C programming language. Both Max/MSP and PureData support “externals”.

Section 5.1 beginning on page 51 provides more information on MaxMSP programming conventions and concepts.

1.4 Objectives

The project proposal form listed the following original objectives:

- Create a software synthesiser that will allow a user to produce music with chiptune timbres
- Recreate a complete set of sound generation (and design) techniques to cover the entire range of timbres produced by the custom chips for which chiptunes were composed
- Design an interface that provides a high level of control over synthesis parameters, without overpowering the user with detail, or abstracting them too much from the underlying synthesis processes
 - Visual approach, drag and drop re-ordering of step sequences
- Prototyped entirely in an environment such as MaxMSP or PureData
- Possible extensions to prototype:
 - Provide control tables for FM synthesis parameters
 - Create a library of presets with “genetic mutation and breeding”
 - Convert to a modular system with custom *externals*,
 - Standalone Audio Unit or VST instrument
- Learn how to create custom graphical user interfaces for software like Max/MSP and PureData.
- Learn how to create custom external objects for Max/MSP and PureData.

Evolution:

Over the course of the project, these objectives were modified slightly to reflect the time constraints of the project module, and to reflect different choices that opened up in light of information discovered during research.

These changes occurred at different stages in time and affected the development of different components. Wherever possible, this report will explain briefly the original thoughts, the reasons for changing, and the final decisions in the appropriate section. An overview of the major changes to the original objectives:

- Time constraints caused the abandonment of
 - Genetic mutation and breeding
 - Drag and drop re-ordering of step sequences

- Graphical interface external
- Also, the "complete set" of techniques was reduced to a small selection
 - This small selection was chosen to maximise the ability of the system to create the desired sound character, rather than a software implementation of certain specific hardware and software techniques
- An external was written, but not a graphical one. The inclusion of this external into the overall system was abandoned, but valuable lessons were learned during development.
- There was great difficulty in organising the vast amount of documentation and paperwork generated during the project. An investigation into the LaTeX document processing engine revealed useful techniques for cross-referencing and chapter numbering.
 - The section on report management covers this in more detail. Explaining how these techniques were recreated in Microsoft Word as much as possible.

1.5 Strategy

This section aims to describe the author's adherence to good practices in project management and software design, to show that the project has been thought through, well structured, and that the author's thought processes on how to combine and synthesise the original ideas that inspired the project are structured and methodical. It explains the justification behind the decision to separate the interface from the synthesis in an effort to avoid getting stuck in an endless cycle of development, where work on one area renders another obsolete. Similarly, this report attempts to break down the structure of the project into discrete sections that make for easy reading.

Throughout the report, the synthesis structure is sometimes referred to as being separate from the interface, but the two are intertwined. Sound chips were configured by software, which contained instructions for controlling the synthesis elements on the chip and provided a specific user interface to control this. When investigating the benefits of a particular historic interface, the synthesis technique that it controls may also be a factor and vice versa.

1.5.1 Design strategy

This section aims to get to the core purpose of the project; that is of achieving the expression of an idea in a composer's head. From the point of view of the user, the overall picture is that a user will have a chiptune idea, use the project, and hear the result at the end much more quickly and conveniently than if the user had attempted to use a variety of older computer systems to compose separate elements. When this scenario is examined closely it becomes apparent that a user will need to interact with an interface to control the synthesis structure in order to hear a result.

The reader will appreciate that there are often many different ways of achieving a given result. When synthesising a sound or waveform, there are many ways of organising a given synthesis structure. Each way will have a different set of parameters for controlling the synthesis aspects. The lists of parameters for a particular synthesis structure are tied to the structure that they control. The components of two synthesis structures could be different, resulting in a different parameter set, but even if the components themselves do not change, the way they are connected could be different and still produce an identical result. The bottom line is that any synthesis structure has a corresponding parameter set.

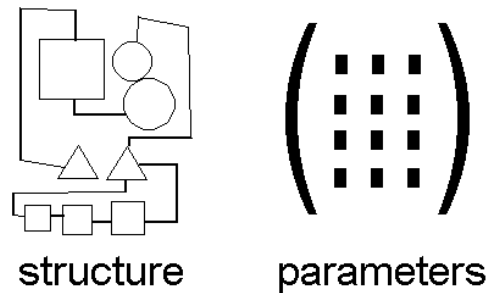


Figure 1.4- Any synthesis structure has a corresponding parameter set

Synthesis structures have two components; the structure, and the controlling parameters, an interface has similar components. An interface will have its appearance, or form, or layout and a mapping that relates *interface elements* onto the *synthesis controlling parameters*.



Figure 1.5 - Each interface maps onto the synthesis parameters

Also, just as different synthesis structures will have a specific set of parameters associated with each structure, each interface mapping is specific to that parameter set. No matter what form, layout, or appearance the interface has, the mapping component is part of the communication of the musical idea into the synthesis structure that will render, or realise, or sculpt the sound.

During the design process, many things could be in flux as different methods are tried out. Time is a valuable commodity that cannot be wasted. Since things are likely to be in flux during the design stage, that different synthesis structures will result in different parameters, and that the interface will manipulate the parameters in order to control the synthesis; it seems logical not to design an interface until the synthesis structure is decided.

To further clarify this point: the model above described the user having an idea, using the interface of the project to control a synthesis structure, and hearing the result. **The user interface, by definition, is the only link between the user's idea and the computer.**

One very important fact to remember is that a “good” (perhaps intuitive) interface need not necessarily model the exact way the synthesis structure behaves.

During the interface design process, it is crucial to have the synthesis structure in place in order to evaluate different solutions. Without a synthesis structure to manipulate, how can one evaluate the practicality of the interface design?

The main problem is that when a change is made to the synthesis structure during development, this will result in a new set of parameters, which will require changes to the interface in order to control this new synthesis structure. In summary: Changes at the synthesis end of the system will require changes to the interface end, and everything in between. It is clear that a **design path of least change is required**.

To summarise the thinking so far:

- A musical idea is communicated to the computer by a user interface.
- The end result is always a sound that can be heard.
- There are many ways to synthesise the same audio result, with different methods having different controlling parameters.
- The user interface maps onto the synthesis structure (comprised of controlling parameters).
- An interface can be seen as an abstraction of the synthesis structure.
- Parameter sets are a reflection of the synthesis structure.
- Minor changes to the synthesis structure will produce a new parameter set
- Interface mappings depend on the parameter set
- Minor changes to the synthesis structure will require a new interface mapping
- Interface form/layout does not (necessarily) depend on the parameter set

It is clear from the above points that changes to the synthesis structure will filter down as far as the *interface mapping*, but that these changes need not affect the actual form or layout of the interface itself. A design path of least change is possible if the interface form is designed last and the synthesis structure is designed first. It will be possible to incrementally solidify the structure of the system instead of having everything in flux.

In closing:

- The sound chip synthesis components are well documented, since the concept was implemented many times for many different computer systems and sound chips.
- It follows that a well defined set of components will have a well defined control data set
- Implementing the synthesis structure will be a matter of finding the most efficient method of realising the required synthesis components in software.
- After the synthesis structure is fully implemented, the author will be free to explore the different interface layouts and mappings, knowing that the control parameters will not change.

Since any interface is an abstraction, and therefore independent of the synthesis structure; research into interface design could be carried out in parallel to the implementation of the synthesis. When the synthesis structure is finally solidified, the interface mapping can be designed to link the two together.

The main benefit of designing the synthesis structure first and the interface last, is that the interface will not have to be redesigned when and if the synthesis structure changes. The drawback of this is that the synthesis structure will have only rudimentary interface components during the design phase.

Once the required synthesis components have been defined, the steps required to produce an interface to control the synthesis should be very clear. The various possible user interface layouts and associated mappings will slot neatly into place over the clearly defined synthesis structures and their associated parameters.

1.5.2 Project management strategy

Many project management techniques are designed to manage the scenario of a large group of people interacting and co-operating, with each person working on a specific part of a larger problem. Project management techniques aim to spread the workload and synchronise the completion of tasks to coincide with the beginning of other tasks.

Although this project has several similarities with the described scenario, several things set it apart from the traditional sense of project management:

- Time constraints (Other modules are taking place at the same time)
- Only one person is working on the project
- Element of learning (Many things must be learned in order to complete the project)

The main difference lies in the element of learning built into the module. That is to say the author may be required to, learn new skills, perform research, presenting a list of achievements in order to complete the project module. Complications in planning may arise where the author is unable to say how long it will take to learn a new skill, or to research a given area in order to find the answer to a specific question before the next task can begin.

These sorts of problems are less likely to occur in a traditional project management process where the individuals working on a project are brought to the project before any work begins, based on the required skills and research knowledge.

Certain project management techniques can still be applied to the project, however. A Work breakdown structure (WBS) is a list that breaks down the implementation of a system into discrete tasks that are to be completed. Some components may rely on the others being complete, whilst other components can be designed in parallel. A Gantt chart is a graphical representation of the WBS with time running along the X-axis and simultaneous tasks stacked up along the Y-axis.

The WBS and Gantt chart are the first things to be constructed before doing any project work, they are the fundamental benchmarks of progress in the project, they should be referred to repeatedly to measure progress, they will keep the project organised and will help ensure that deadlines are met. They are also revised often, with deadlines being shifted if things go well, and new tasks being added if things go wrong.

In the early stages, only the project objectives were clearly defined. Whilst certain objectives could be counted as tasks in themselves, some high-level system design was required in order to derive a more detailed WBS. This high level design consisted of creating a "System Model" that outlines the required system components and how they will interact.

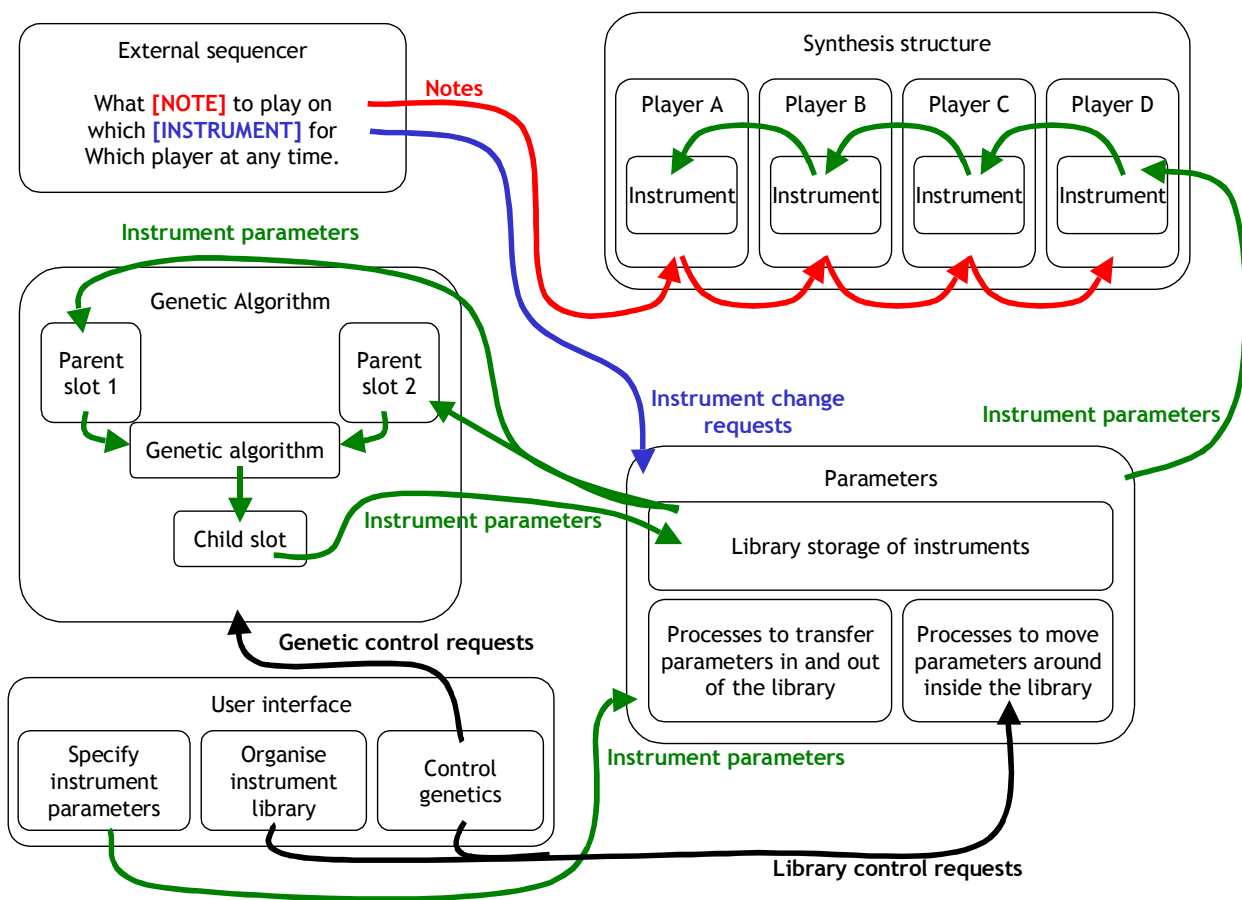


Figure 1.6 - Original system model

This system model closely reflects the structure of the final system. The only component that was not implemented is the genetic algorithm component.

From the system model, a more detailed WBS and a Gantt chart was constructed. Each component in the system model became a “task” on the WBS and Gantt chart. A rough timescale was drawn up for each component and used to create a Gantt chart. This timescale was influenced by the deadlines for certain project documentation presented in the Final Year Project Guide, comments from the author’s project supervisor, and deadlines for other modules.

An overview of the Gantt chart is presented below. It is difficult to discern details from this chart due to the compressed time scale; *a full version is available in the appendices*. Figure 1.7 is intended to show the general spread of activity and overlapping nature of development.

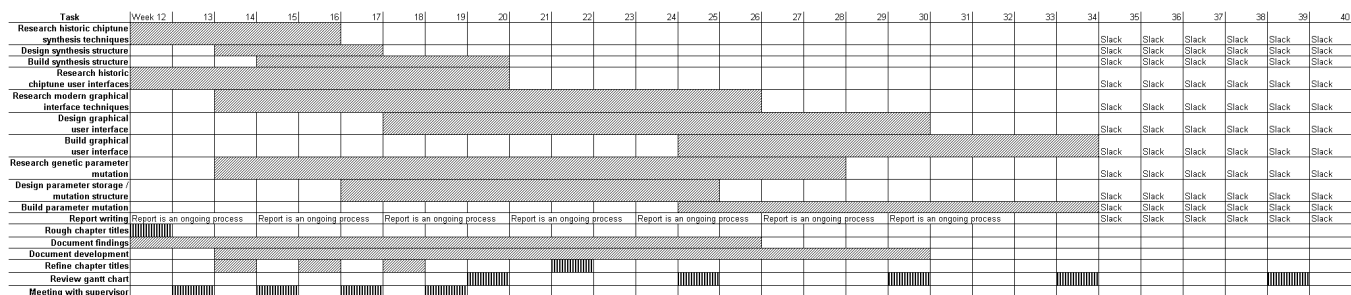


Figure 1.7 - Gantt chart overview

The original Gantt chart breaks the work up into the components identified in the system model: Synthesis, Interface, and Genetics. As time progressed, and research uncovered new avenues to explore, the original plan outlined in the Gantt chart became less relevant. The Gantt chart concentrates on the implementation of the system components; no time was allocated for the author to develop the efficiency of the components although this task was indeed carried out (see Section 5.7 beginning on page 110). The Gantt chart also omits a major decision covered in Section 2.1 (page 25), the choice of development environment (MaxMSP or PureData). This decision was especially difficult since the initial experimental prototype was developed in PureData. Porting the prototype functionality to another (albeit quite similar) environment was an initially daunting prospect.

Time constraints meant that the original strategy required modification. For example; the original intent was for the synthesiser to emulate the sound character of a wide range of existing systems, but after the research uncovered so many different types of system each with their own unique quirks and desirable characteristics, the decision was made to reduce the complexity.

More details of the design strategy for each system component and how they evolved are presented in separate chapters. Full versions of the Gantt charts and WBS charts can be found in the appendices.

1.5.3 Software development strategy

Whilst the previous sections have concentrated on the strategies for organising the thought processes, and time management involved with the project, this section concentrates on the strategy adopted during the implementation phase.

Some projects depend on applying established techniques alongside personal experience in order to solve problems. The design and specification of the solution is carefully planned out, greatly assisted by using this previous experience in order to sidestep the problems that are likely to be encountered. The specifications produced with foreknowledge of previous systems are produced much more quickly than the specifications that must be produced iteratively, solving each problem as it is encountered. Time restrictions, and lack of previous experience prevent the author from producing a specification in this way.

An incremental development strategy was considered for the project. With regard to this method, Sommerville (2004) explains “system requirements ALWAYS evolve in the course of a project so process iteration where earlier stages are reworked is always part of the process for large systems.” This implies that any specifications, plans, and designs that are made will always be revised over the course of a large project. This concept is closely related to the incremental development process, whereby the system is built up in stages.

Sommerville explains this further by saying “Incremental delivery [involves] breaking the development and delivery down into increments with each increment delivering part of the required functionality” The figure below explains this concept graphically.

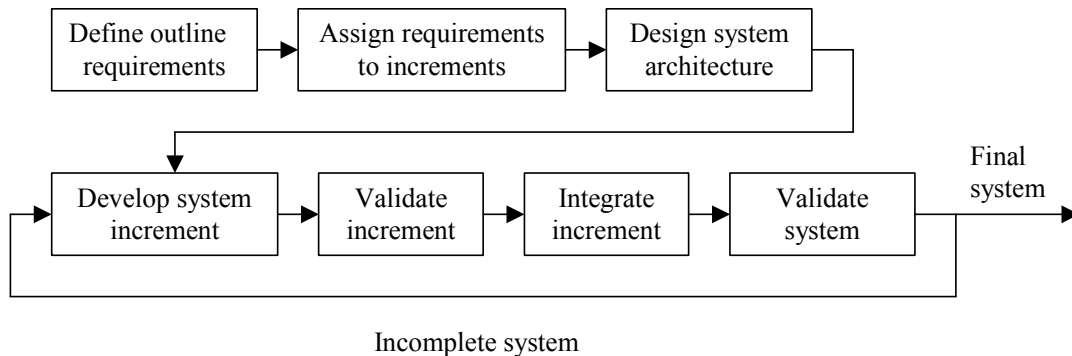


Figure 1.8 - Incremental delivery (Sommerville, 2001)

Incremental delivery is especially suited to a modular system. By breaking the system architecture up into discrete modules (as in the system model outlined in Figure 1.6 on page 12) each run of the incremental delivery method can be devoted to improving the design of a separate module, with a further run of the incremental delivery method to ensure that the entire system structure is working co-operatively towards meeting the system requirements. Earlier versions of the system can be used as prototypes for later increments.

A modular system is built in stages, starting from a simple structure and gradually adding more complex functionality. This reduces the risk of overall project failure. Building the system in stages means that even if the more complex stages are not complete by the deadline, the simpler stages will be functional at the project presentation.

If the system was designed as one big block of functionality: early decisions, made without a complete understanding of how the final system will behave, may have a negative effect. Choosing a modular approach makes any decisions made at an early stage have less of a global impact if the overall system is made from modules that can change and adapt.

Choosing to implement a modular system will make the development process easier to fit into incremental delivery method, since modules are built in stages, it also makes it possible to have some components implemented in different ways (see Section 2.1 for more on this).

A modular approach could make it easier to extend the project by converting graphically programmed modules to “externals”. A modular approach would also force “patch parameter data” to be exchanged between modules it may also make the design of the “genetic patch mutation” module easier.

It is important to note that in addition to the incremental and modular approaches discussed, the spiral model of development is particularly relevant. The spiral model is useful for projects that involve a great deal of experimentation since it allows for hopping between development stages to be started as and when they are required.

1.5.4 Report structure management

In the early stages of the project, the proposal form (see appendices) explained that the author's approach to documentation would be an ongoing process, the intent being to produce formal documentation at each stage of development to reduce the need for a concentrated documentation effort at the end of the project. One reason for choosing this approach is to avoid the potential hazard of identifying important information at an early stage but forgetting a great deal of this by the time the documentation process begins.

The author discovered the drawbacks of this approach quickly, abandoning it in favour of keeping a brief, informal documentation folder with rough notes and ideas that would later be formalised:

There are several problems with keeping formal documentation written at every stage of the project development:

- A vast amount of documentation soon piles up; most of it outdated the moment a new design decision is made.
 - The outdated documentation may still contain useful concepts, so none of it can be completely discarded.
 - A “concentrated documentation effort” is not avoided at all, since all of the outdated documentation must still be scoured for potentially useful material when the final report is compiled.
- Restructuring the existing formal documentation to present the core ideas in a logical manner becomes difficult.
 - Each of the outdated formal documentation files relied on the core ideas but expressed them in different ways.
 - Each revision seemed perfectly logical in itself, but the order in which the core ideas were presented was often different between versions.

A notable amount of time was spent trying to structure the report layout in terms of the order that the core ideas are presented. A great deal of time was spent trying to use as much of the existing material as possible, combining the different approaches to explaining the core ideas, without the result ending up with paragraphs repeating core ideas over and over in slightly different ways.

The author set up a dual-head monitor system to view many pages of the documentation at once, aiding the process of combining the different versions of the documentation.

It was suggested that the LaTeX document preparation system be used to organise the final report. Rather than learning the LaTeX markup, the author investigated possible solution of using the LyX software (Lyx.org, 2007) to rapidly organise the text in a logical manner, making the process of cross-referencing and bibliographical referencing easier.

Problems were encountered with postscript and bitmap graphics on the author's system, with simple documents taking many minutes to preview.

Given the extent of graphics use in the report (for example, the large number of screenshots in the implementation chapter), the author returned to Microsoft Word as the documentation environment. Microsoft Word has quite capable cross-referencing and document outlining tools for structuring chapter, section, and sub-section headings. The documentation approach was revised to involve writing smaller numbered sections instead of large sprawling chapters, re-ordering and cross-referencing the content in these smaller sections instead of repeating core ideas over and over.

With reference to printing, the dual-head monitor system mentioned previously was used to enable proofreading and checking of references to other pages with ease. Also, the decision to print certain pages in colour separately was made to enhance the reader's understanding of certain key concepts by the use of colour coding.

Several past final year project reports kept in the university library were also examined for their structure.

1.5.5 Testing strategy

As discussed in section 1.5.3 , a modular incremental development approach was considered. The gradual application of the spiral model, however; allowed more rapid development due to the freedom afforded by hopping between different stages of developing, testing, and assessing results during each increment stage.

Verification and Validation.

Section 1.5.3 also discussed the author's lack of experience and the time restrictions placed on the final year project module prevents true validation procedures from being carried out. The experimental development process presents a problem in creating a firm specification that can be compared against the end product to validate the solution. This means that the project objectives can be validated, but not necessarily the software objectives and specification.

The software can be "verified" however to confirm that the system is built well, and operates on the users input in a predictable manner. It is for this reason that the testing strategy will focus on the user interface.

The numeric user interface objects that MaxMSP provides have the facility to be bounded by upper and lower limits. This facility can be used in order to prevent a user from entering a value that will cause a lower level process to malfunction or communicate incompatible data to other processes as a result. It should be noted that this method relies on the MaxMSP validation routines, which are outside of the author's control. If the author had written a custom user interface, it would be possible to run tests on the input validation code too.

White and black box testing

Two commonly quoted testing categories often applied to software systems are white and black box testing. Black box testing, focuses on confirming the functionality is as expected, without knowledge of how the functionality is achieved and will fully test the functional requirements. White box testing involves using the known structure of the program to fully

explore the entire structure of the program. White box testing tries to use every combination of inputs in order to make sure that every branch, every conditional test, and every loop runs at least once and produces an output which can be tested for its conformance to the specification.

An exhaustive white box test is not particularly well suited to the final year project since there is a limited amount of time available, and as explained before the software component specification is not rigorously defined.

Multiple test platforms

The decision to test on more than one computer was made due to the potential benefits of discovering how the system behaves in different situations. It will be possible to test the system on Apple and Microsoft operating systems due to the fact that both MaxMSP and PureData are available on both architectures.

The two systems used for testing were:

1. Acer laptop:

Intel PentiumM based laptop running WindowsXP pro and MaxMSP 4.6

2. Apple G5 Powermac:

Motorola PowerPC based machine running OS10.4 and MaxMSP 4.5

Individual modules were tested on the Acer laptop development platform, whilst integrated tests of the entire were performed on both the Acer laptop and the Apple Powermac.

Some cross-platform problems were encountered with regards to the display of the user interface between the Apple and Acer machines. This was attributed to differences in font sizes, and possibly slight differences between backward compatibility of the differing MaxMSP versions.

Chosen test strategy

The chosen test strategy was therefore based on a black box approach. This would be a limited test concentrating testing each input of each major component in the system, and comparing the expected output to the actual output. The end result was a list of sensible upper and lower limits for inputs that can be enforced at the user interface by the MaxMSP numeric input objects. Some of these limits are pre-determined by external factors, inputs related to MIDI are constrained within 7 or 14bit integer values as a direct result of the MIDI specification.

During the development stage, the user interface limits were not imposed since the major components themselves were under development. Some limits were set as development progressed, but the final test strategy of testing every input of every component was fully carried out after the author was confident that the major development effort was over.

The test results are not summarised in one place, testing of individual modules are discussed in the implementation section (Chapter 5 , page 49) in the form of testing notes accompanying the description of each component in the system. Discussion of the results of the integrated testing can also be found in these notes.

1.6 A brief overview of MIDI

The Musical Instrument Digital Interface (MIDI) is a system for interfacing control signals between (among many other devices) musical trigger interfaces, computers, and synthesisers. This section does not attempt to describe the full MIDI 1.0 specification (Midi.org, 2007) only some of the most “important” qualities (in terms of this project) are summarised here.

The term “MIDI” implies two things:

- The physical interface standard of 5-pin DIN connectors carrying currents between hardware
- The communication protocol that the data flowing between physical connectors must obey in order to comply with the MIDI standard

In terms of the physical connection, the original MIDI specification details a system whereby multiple hardware devices can be connected together to enable a collection of electronic music devices to communicate and synchronise their musical output. The MIDI communication protocol is concerned with parameters and triggers, and therefore communicates things like:

- Synthesis parameters and the nature of any change to the sound character over time
- Which musical note to play, and how loud the note should sound. This is achieved by “note on” and “note off” messages.

An example MIDI setup might be a “controller keyboard” similar in design to a piano keyboard with a number of triggers that create unique MIDI messages. These messages can be interpreted by a MIDI compatible synthesiser module in order to produce sounds that are triggered by pressing different keys on the controller keyboard. Different MIDI controllers exist such as wind controllers that are designed in the shape of a wind instrument; the position of the trigger keys in a similar position to that of a saxophone or clarinet.

The MIDI specification was designed as an interface between hardware devices in the “real world”. In recent times, however; software synthesis has become popular and the concept of a “virtual” MIDI connection is not uncommon. A virtual connection, through which MIDI messages such as note triggers and synthesis parameter values, is often set up to control the software synthesiser as if it were an external hardware device connected with a physical MIDI cable.

It is the intention of this project to use the MIDI protocol controlling a software synthesiser by MIDI messages generated and received internally through “virtual” MIDI cables. This ensures compatibility with modern composition software that uses the MIDI protocol to sequence musical notes. The adherence to the MIDI protocol - as opposed to another internal communication system between sequencers and synthesisers such as Steinberg’s VST (Virtual Studio Technology) protocol - also enables the possibility that the synthesiser could be used in a live performance situation using a MIDI controller.

1.7 Trackers vs. MIDI sequencers

This section highlights the differences in composition techniques between trackers and MIDI sequencers, the problems of trying to transport features commonly present in tracker interfaces to the project, and the difficulty of maintaining a solution that will enable the user to use the project to compose using a MIDI sequencer such as Cubase 5 (Steinberg, 2001), a MIDI Tracker such as MIDITracker (Rf1.net, 2006), or as a live performance synthesiser.

There is a fundamental difference between trackers and MIDI sequencers. This is evident in the mindset of the composers, and to some extent the accepted interface techniques:

A composer using a tracker is likely to input the music note by note (using a computer keyboard) to produce the final piece, programming every detail from the ground up. A composer using a MIDI sequencer is more likely to perform or play the music into to the computer (using a MIDI controller), modifying the recorded data to produce the final piece.

The following figures show how a four part musical phrase is represented in three different ways:

Figure 1.9: Traditional music notation

Figure 1.10: Cubase 5 key editor

Figure 1.11: MIDITracker interface



The image shows a musical score for a four-part phrase, labeled 'Midi 1' through 'Midi 4'. Each part is written on a five-line staff in treble clef with a 2/4 time signature. The notation includes various note values, rests, and accidentals. The first staff (Midi 1) starts with a '1' above the first measure. The second staff (Midi 2) starts with an '8' above the first measure. The third staff (Midi 3) starts with a '4' above the first measure. The fourth staff (Midi 4) starts with an '8' above the first measure. The score is divided into two measures by a vertical bar line.

Figure 1.9 - Four part phrase shown as music notation

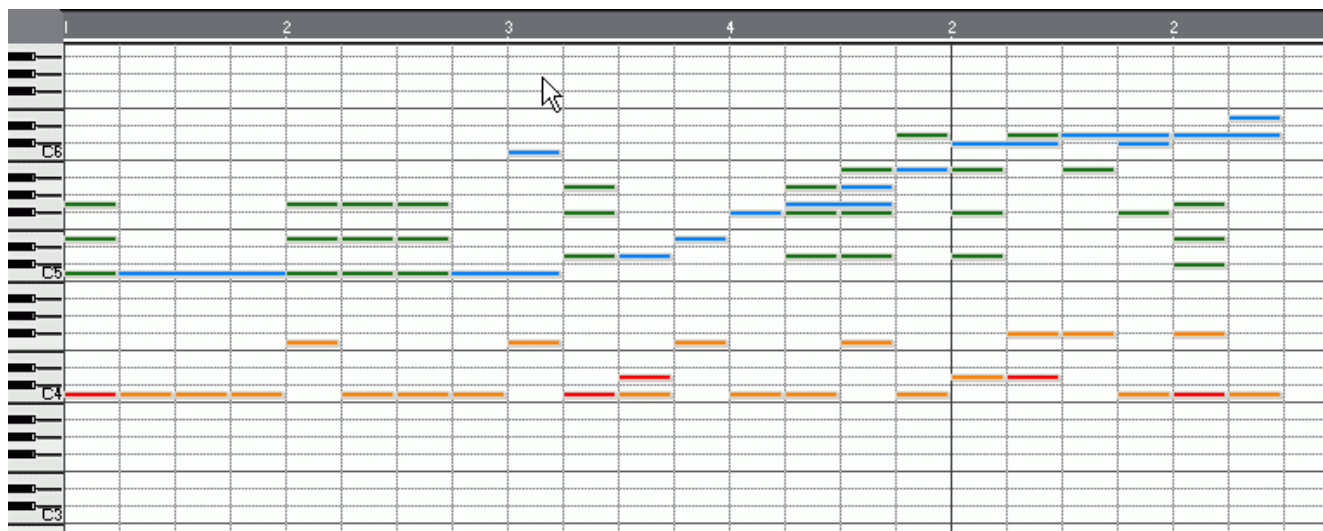


Figure 1.10 - Four part phrase shown in Cubase 5 “key editor”

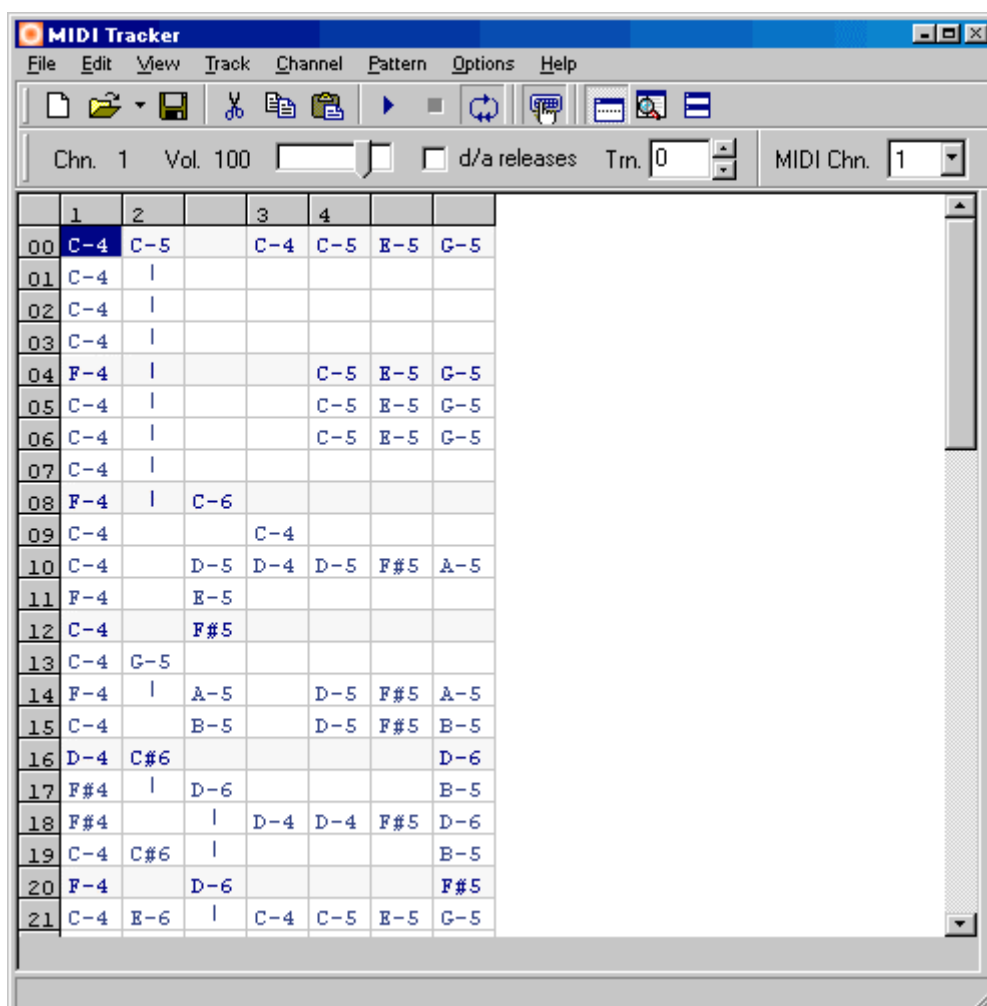


Figure 1.11 - Four part phrase shown in MIDITracker

Chiptune music, by definition, is “programmed” rather than played; the performer of the composition is a piece of electronic hardware.

For this project, a synthesiser is to be constructed which could be used in a number of contexts, either as a live performance or with an external sequencer. In either case, the control will be split. Note triggers are stored in the MIDI sequencer or played on a MIDI instrument, but the act of designing a sound takes place by using the project synthesiser. In order for this not to be confusing to the user, it must be very clear where certain actions are to take place.

In tracker software the sequencer and synthesis control interface was often combined into the same piece of software; instructions for programming the sound character in an “instrument” could overlap with the programming of the notes, since the tracker is the only piece of software being used to control the sound chip, instructions sent to the chip can be displayed and edited in any context within the tracker software. As explained before; the “hyper-sequencer” nature of trackers meant that the synthesis parameters could be altered *whilst notes were being played*, as part of the sound character. Whilst this is very powerful, it is also potentially confusing.

Although this project is primarily concerned with implementing a synthesiser, it is important to consider the possibility that the project will be used with sequencers and as a live performance instrument. The reader should appreciate that it may be difficult to implement a single interface to control this “hyper-sequenced” behaviour that is effective in both a live performance context and with a MIDI sequencer.

An obvious solution to this problem would be to provide a sequencer as part of the project, but this would force a user into a certain way of working. It can be said that part of what makes a “bad” interface is forcing a user into a particular way of working, whereas a “good” interface might suggest certain ways but is generally adaptable to the way of working that the user prefers, *not the other way round*.

Having said this: a system called ReViSiT (formerly called VSTrack) developed at Trinity College in Dublin, is a system that wraps a tracker interface into a VST plugin very successfully.

“By importing a tracker interface and architecture into a sequencer VSTrack is a tracker coded as a plug-in to a sequencer, integrating with the sequencer application, receiving synchronisation information directly from the host, and returning multiple channels of high-quality ‘tracked’ audio. The net result is that the composer can retain the familiar sequenced tracks and “multi-tracking” audio utilities of the host, while simultaneously benefiting from the advantages offered by trackers” (Nash, 2004)

The end result is a liberating combination of composition interfaces that give the user a choice in their working methods, the synchronisation between the multi-track host sequencer and the plugin tracker afforded by the VST protocol means that the user can switch between working methods at any point. Bringing the focus back to interfaces, the reader should note that the VST protocol used here presents an alternative way to communicate note triggers internally as mentioned in the previous section on MIDI.

In summary: it is important to divorce the design process from trackers and their ideas, and get to the bigger picture. The overall issue is of expressing an idea in a composer's head, through to sound hitting the ear of the listener. The characteristics of the historic synthesis (sequenced parameters) are desirable, whilst the limitations of historic interfaces are not.

1.8 Layered systems

The OSI 7 layer model is used in computer network communications theory to describe the various layers of software and hardware that are involved in a computer network.

Layer	Function
1 Application	Network process to application
2 Presentation	Data representation and encryption
3 Session	Inter-host communication
4 Transport	End-to-end connections and reliability (TCP)
5 Network	Path determination and logical addressing (IP)
6 Data link	Physical addressing (MAC and LLC)
7 Physical	Media, signal and binary transmission

Table 1.2 - OSI seven layer model (Tolhurst, 1988)

In Table 1.2, the physical connection of currents flowing along conducting cables is represented by Layer 7 which is a hardware layer. Layer 6 represents the systems on board a network interface adapter, another hardware layer. Layers 5-1 are mostly software layers, with Layer 1 being the application layer. Layer 1 typically presents the user interface that will enable the user to control and use the system. This does not necessarily mean the application layer will present a way to specify or alter the behaviour of all layers beneath it, merely that all layers are in some way sparked into action by the signals originating at the application layer.

Layers 2-5 are generally software systems that the user has little control over. They are sub-systems that are designed to operate in accordance to a strict protocol.

The idea of configuring low-level systems from the application layer is extremely important in the context of this project. The computer systems that this project is concerned with commonly consisted of a microprocessor controlled sound chip. This situation can be divided into layers of functionality too. If the concept of layers from the OSI model were applied to a generic microprocessor controlled computer with a sound chip running a tracker, the layers might be labeled something like Table 1.3:

Layer	Function
1 Application	Design sounds, compose music, alter parameters (Tracker)
2 Microprocessor	Control and configure devices lower in the chain
3 Timing	Provide time base for oscillations/musical rhythms (Clock and Interrupts)
4 Device communication	Facilitate exchange between devices (Data bus)
5 Sound chip registers	Provide temporary storage of synthesis parameters
6 Oscillators/envelopes	Produce oscillations and amplitude variations
7 DAC output and beyond	Conversion of digitised values to analogue, and transmission of signals to amplifier

Table 1.3 - OSI layer model applied to a microprocessor controlled sound chip

In Table 1.3, the situation presented by the computer systems is quite different to that of network communication; here Layer 1 is the only layer that is represented by software and yet the behaviour of the Layers below, right down to Layer 5, are accessible for complete configuration.

The programmer of application software is presented at Layer 1 with methods of controlling the behaviour of multiple layers by using the microprocessor and other hardware of the computer. The programmer of a tracker is therefore able to control and configure the actions to take on each layer.

In terms of controlling sound synthesis, this allows great flexibility for a programmer to use the sound chip as a co-processor alongside the CPU, expanding the synthesis structure present in the sound chip, expanding it to take on the other hardware present in the microcomputer. In systems such as these, the levels beneath the application layer are structured in such a way to provide control of low-level hardware. The data to be communicated to the sound chip registers, the rate at which hardware interrupts will occur and the instructions to be carried out in the interrupt service routines for modifying the data held in the sound chip registers can all be controlled from the application layer.

The application layer presents a way to handle low-level data, at a high level, taking care of the required actions to achieve the desired result. The important thing in the context of chiptune programming is it also allows the user to specify very low level things if desired.

Changing subject a little: The OSI layer model is a model for bi-directional network communications; in the original OSI layer context a signal occurring at the lowest layer is filtered back up the chain to the application layer thus allowing two users to communicate at the application level.

In the situation of a microprocessor controlled sound chip, all the control signals are flowing downward toward the sound chip. Usually the only feedback to the user is from the DAC output to the user's ear, any user interface feedback messages that communicate what the sound chip is doing are implied rather than actual.

This is to say that the sound chips are strictly output devices when it comes to their synthesis. The control registers on the chips were often "Write Only", meaning data could be

poked into them, but that it was never intended to be read back again to report the status of the internal synthesis for example.

The idea that information about the status of the synthesis is displayed in the user interface of these historic systems reinforces the idea that control is strictly downward: the user controls the synthesiser, right down to the values in the registers of the chip.

The sheer flexibility of these historic microcomputer systems presents a problem when trying to implement a similar type of system in MaxMSP. The task is made simpler when remembering that it is the sound character that is important: The project is not to present the complete structure of a microcomputer, with all its possibilities for control. This project specification is to allow a user to create chiptunes, which have very specific, definable characteristics.

In the coming chapters, the following topics will be covered:

- In Chapters 2 and 3 the various components involved in chiptune synthesis are researched, analysed.
- These components are assessed for their importance in contributing to the sound character of chiptunes in chapter 4.
- Finally, in Chapter 5, this catalogue of components is reduced to form the final set of components which are implemented in MaxMSP.

Chapter 2 Interface considerations

This chapter discusses the concept of a User interface and a Machine interface. The main purpose of this chapter is to introduce the reader to the user interfaces of trackers, and to identify interface concepts that influenced the author during the research. The chapter also discusses the way sounds might be stored for possible access by a genetic algorithm. This chapter also analyses the differences between two methods of constructing a sound library popular with trackers, and with hardware and software synthesisers.

Discussion of the actual implementation of the user interface is not in this section, for this, see section 5.4 beginning on page 88.

2.1 Choosing the development environment

In section 1.3 (on page 5), two systems (PureData, and MaxMSP) were discussed as possible development environments. Despite their similarities, PureData and Max/MSP are not identical, neither in operational use or internal design. For example, MaxMSP provides many advanced graphical interface objects as standard; whilst the interface objects provided in PureData are limited in number and in configurability. Some of the interface objects provided with Max/MSP are “externals”, but they are ones that come with the software bundle rather than third party solutions. This gives Max/MSP much more scope for designing custom interfaces.

If only “native” user interface objects are used, the choice of environment is strongly linked to the ease with which an interface can be developed at a later stage. To some extent, this is also true if custom self-built “externals” are developed.

Some research was made into the various capabilities of each piece of software (Max/MSP and PureData), also the possibilities of interfacing to external software via “externals” or otherwise.

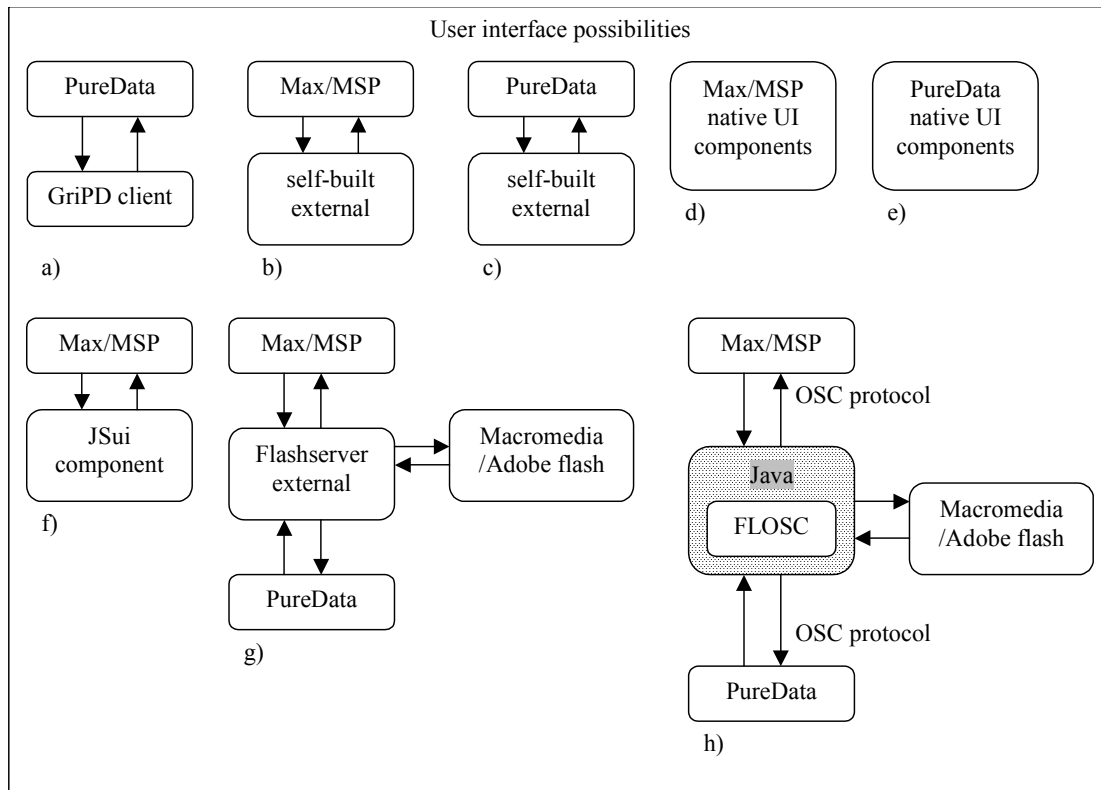


Figure 2.1 - Possibilities for user interface design

Figure 2.1 is included to show that various possibilities for creating the user interface were considered. Each scenario is assigned a letter, most should be self explanatory; part “f”, for example, shows that Max/MSP could be used in conjunction with the JSui external, while part “g” shows that the Flashserver external exists for both Max/MSP and PureData.

In Figure 2.1, part “a” shows the possibility of designing the synthesis structure in PureData, using GriPD to create a separate user interface. “GriPD is a cross-platform extension to Miller Puckette’s *Pure Data* software that allows one to design custom graphical user interfaces for PD patches. GriPD is *not* a replacement for the PD Tcl/Tk GUI, but instead is intended to allow one to create a front end to a PD patch. The concept is to create your PD patch normally and then your GUI using GriPD” (Sarlo, 2006).

Part “g” of Figure 2.1 shows the possibility of designing the user interface using the “Flash Server external” for Max/MSP. “The flashserver external for Max/MSP allows for bidirectional communication between Max/MSP 4.5 or later and Macromedia Flash 5.0 or later” (Matthes, 2005) It is an external that will enable the benefits of rapid development for vector-based graphic interfaces in Adobe (formerly Macromedia) Shockwave Flash.

Part “h” of Figure 2.1 shows the possibility of using FLOSC (which stands for “Flash Open Sound Control”). Instead of the data being directly transferred between flash and the Max/MSP or PureData patch via custom “external”, the data is transmitted via the OSC protocol to a Flash/OSC server written in Java.

2.1.1 Environment decision

Choosing the development environment before embarking on the project was a difficult decision to make. The author was concerned about choosing the “wrong” environment, which could limit interface design potential at a later stage.

The author is more familiar with PureData as a development environment over MaxMSP. Choosing PureData as the development environment will speed up the implementation of the synthesis structure. The author is aware of the more extensive and powerful interface components that are part of MaxMSP, choosing MaxMSP as the development environment may hamper initial progress as the author would be forced to recreate the functionality of the prototype whilst navigating the differences between PureData.

Choosing MaxMSP would make the implementation of the desired interface much easier however. The danger in choosing PureData is that it may be too difficult to develop a satisfactory user interface. There is, of course, an additional danger that MaxMSP may not be best for designing interfaces after all.

The bottom line is that the chosen environment should make early exploratory development easy, whilst not limiting the interface design process in the later stages.

MaxMSP was chosen as the development environment for two reasons:

1. The number of options that involve MaxMSP presented in Figure 2.1 (parts a-h) greatly outnumber those that involve PureData.
2. The native UI components in MaxMSP present fewer limitations than the native UI components of PureData

The strategy outlined in Section 1.5.1 , was to develop the synthesis structure first, and the interface last. The decision to choose MaxMSP is reinforced by the idea that if, after the synthesis structure is complete, the native UI components of MaxMSP exhibit a limitation that prevents further progress; MaxMSP supports plenty of alternative interface development routes.

2.2 Historic interfaces

Since this project is an attempt to implement a historic system of music synthesis using modern tools. It is logical then, to look at the ideas presented within the interfaces of these historic systems. The final solution will be a combination of desirable modern and historic ideas.

As described in Section 1.7, elements of the synthesis technique and the software used to configure the synthesis are closely linked. Care must be taken when examining historic interfaces for inspiration. With reference to the hyper-sequencing of trackers (where a single note can trigger sub-sequences of oscillator waveforms, and modulation parameter sequence); the author is very aware of how hard it is for new users to get to grips with these multi-layered ideas presented in trackers. One of the project objectives is to define an alternative interface to the traditional tracker presentation of sound design tools. This is extremely difficult since chiptunes are so programmed in nature. It is hard to think of different ways to do the same thing.

The task of examining historic interfaces is therefore carried out with the intention of avoiding the pitfall of bringing entire interface philosophies to the project and forcing the author into certain interface choices. The intention is to bring certain useful and elegant interface techniques to the project, based on ideas from the past and to combine these with modern interface components available in MaxMSP.

2.2.1 Historical sound design interface influences

Magic Synth V2.5 (Seeman, 1997)

This program for the Atari uses hexadecimal values to specify the desired sequence of waveforms that will occur when a note is triggered. This re-iterates the fact that tracker software provides a way to configure the synthesis parameters whilst a note is being played. This method of waveform sequencing is a particular characteristic of chiptunes which is used in order to generate a more interesting sound character from a limited palette of available waveforms.

The reader should note the names of some of these waveforms in Figure 2.2, particularly “Buzzer” and “Mod SID chan+/-1” which will be explained later in the synthesis section. Note also the “silence” waveform, when used in alternation with a non-silent waveform this can be used for to create stuttering effects.

With reference to the programmed nature of chiptunes, the line “E: Jump position” is not actually a waveform identifier at all, but rather an instruction identifier similar to the assembly code “JMP”, representing the instruction for the waveform sequence to jump to an earlier position. This allows a loop to be created, repeating a sequence of waveform changes over and over.



Figure 2.2 - Magicsynth screenshot (Hexadecimal tabular wave sequences with wavetypes listed top)

Cyber tracker (Laustsen, 2001)

This program for the Commodore 64 specifies the desired waveform sequence (and other parameter sequences) in terms of a graphical envelope. Figure 2.3 shows the textual representations of the data described by the envelope, presented to the right of the graphical envelope.



Figure 2.3 - Cybertracker screenshot (Graphical envelope controlling waveform and amplitude sequences)

MaxYMiser (Morris, 2006)

This program for the Atari has a very powerful interface, but one that forces a user into a strict way of working. This too relies on textual sequences of hexadecimal values to specify parameters. The interface is very compact; one very useful feature is the modulation matrix (middle right hand side) showing which modulations (portamento, arpeggio, vibrato, etc.) are enabled for a given instrument. In other tracker interfaces, these modulation characteristics are often specified elsewhere as hexadecimal flags which are a more verbose and less readable form.

If Figure 2.4 is examined closely, the method by which the pitch of an oscillator is specified in a sequence can be seen.

A modulation matrix shows (by a bright square in the appropriate slot) that the square wave oscillator is to have its “fixed frequency” modulated. Below the matrix, a row of values shows the “FIX” value as “17”. This is telling us that sequence 17 is to be used to drive the “fixed frequency”.

In the lower right corner, sequence 17 is displayed; lines 00 to 03 are displayed and read the values 03,04,04,05. At the very bottom right of the screen are the words “Length 08” and “Repeat 00”. This informs us that sequence 17 is 9 values long, and will jump back to line 00 to repeat the sequence when it reaches the last line.

This is exactly the sort of potentially confusing interface that the author has in mind when talking about the difficulty new users experience when faced with composition in trackers.



Figure 2.4 - MaxYMiser screenshot (Textual parameter sequences in lower right and modulation matrix at middle right)

YM Tracker (Vidovic, 1992)

This program presents an alternative to specifying the frequency of an oscillator in terms of a looping sequence. Here, the sequence of values is represented graphically as a series of discrete points that can be manipulated with the mouse.



Figure 2.5 - YMTracker screenshot (Graphical step sequences controlling frequency and amplitude)

Designing arpeggios in this manner by clicking the mouse may be a process of trial and error in order to find the correct mouse position to achieve the desired semitone steps. “YM tracker” uses a dedicated arpeggio interface where semitone steps are specified in terms of hexadecimal values.

The envelope types for tremolo and vibrato types are represented graphically, but specified numerically.



Figure 2.6 - YMTracker screenshot (Hexadecimal vibrato, tremolo, and arpeggio sequences)

Abyss's Higher eXperience (AHX) (Wodok, 1998)

This is a program for the Commodore Amiga that uses entirely numerical and textual parameters to configure an instrument's parameters. The editor shown in Figure 2.7 is very powerful but crams everything in on one screen.

It is worth mentioning the concept of dynamic and static values where a static value could be defined in this context to be a particular waveform, or a particular amplitude value. Again the concept of a tracker being able to alter synthesis parameters whilst a note is being played is relevant here.

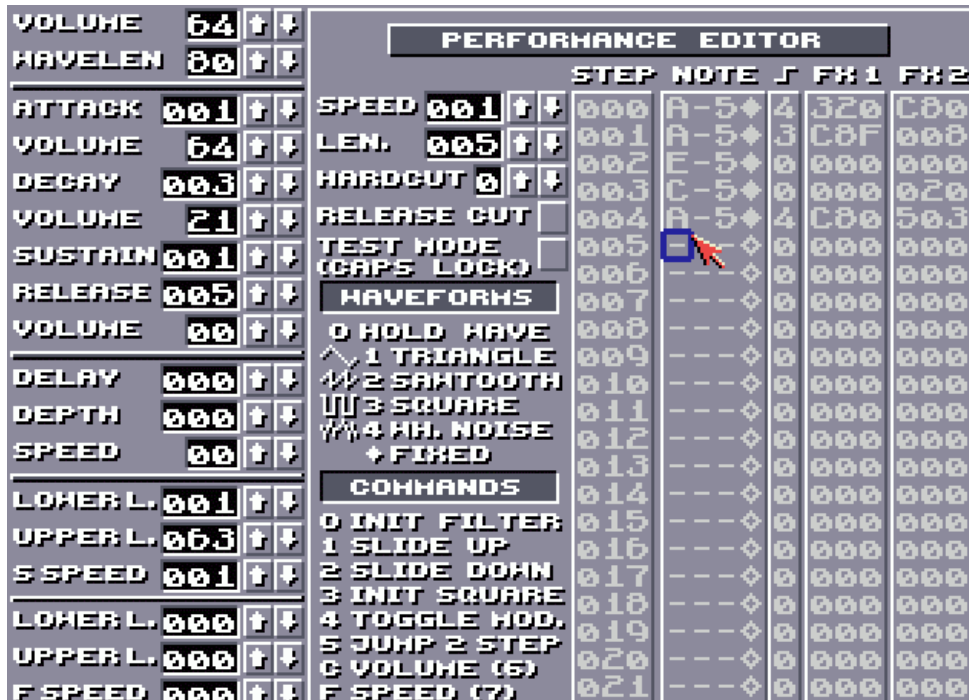


Figure 2.7 - AHX screenshot (Sound editor showing performance editor)

The “performance editor” on the right hand side of Figure 2.7 shows instructions to play a fixed sequence of pitches: “A,A,E,C,A” all in the 5th octave. The waveform sequence is “white noise, squarewave, last waveform played, last waveform played, last waveform played, white noise”. However, the “FX” columns indicate that the squarewave duty cycle should be initialised with a value of 0x20 (decimal 32, indicating a 50% duty cycle) and that the amplitude should be 0x80 (decimal 128, indicating a 50% amplitude) that the low pass filter should be engaged, but be modulated between a cut-off value of 8 and 0x20 (decimal 32). Upon reaching the last line, the sequence should jump back to line 3, where the waveform should remain on “last waveform played” (white noise).

This interface is one of the most powerful the author has come across in terms of chiptune synthesis. An enormous amount of information about the nature of the sound character is described in just 5 lines of hexadecimal values. But it is a rare individual who would be able to interpret these values for the instructions they represent without spending a great deal of time deciphering them. It takes time to learn what each code represents, but once this is done the user can glean a large amount of knowledge about a sound from a single glance.

Musicline editor (Carehag & Fredrickson., 1995)

This is a wavetable synthesiser for the Commodore Amiga whose configuration possibilities are comparable to AHX, but with a far more thought out interface. The use of a strip of radio buttons running vertically down the middle of the screen to call the desired parameter configuration interface into view is of particular interest. In Figure 2.8 an arpeggio is being configured. The section in the upper right is similar to the AHX performance editor in Figure 2.7 where the dynamic element of the sound character can be specified as a sequence of discrete note values.

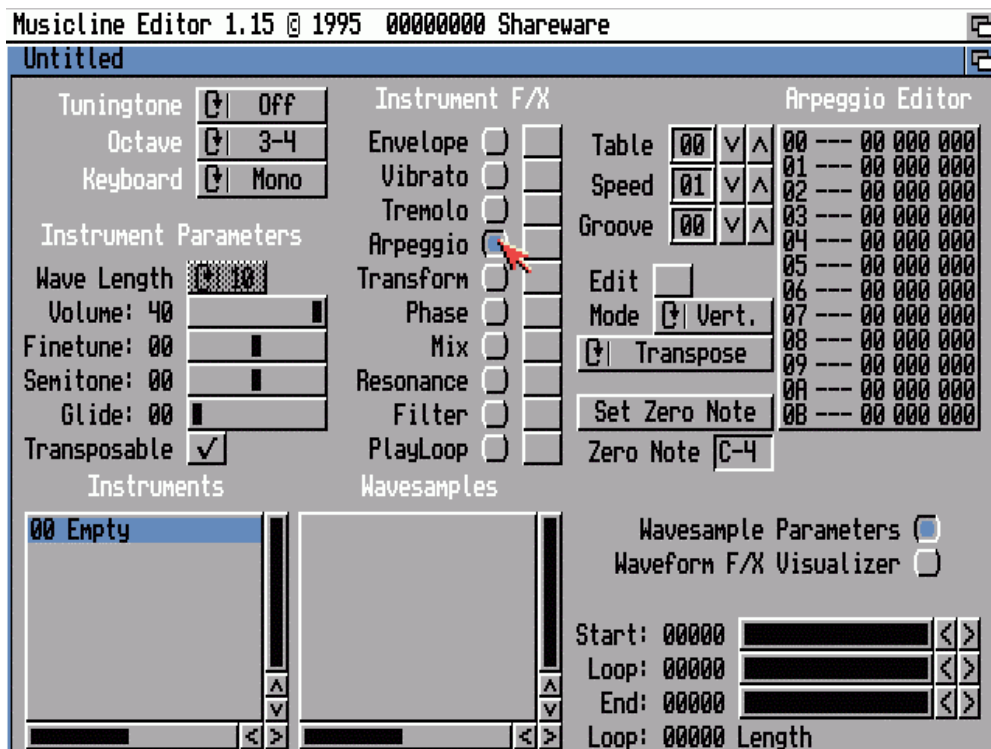


Figure 2.8 - Musicline Editor screenshot (Configuring an arpeggio)

Another really useful feature of “MusicLine” is the “waveform f/x visualiser” as shown in Figure 2.9. This is an oscilloscope that allows the user to view the effect that the configured parameters are having on the wavetable data.

This arrangement implies a two-way connection between the interface components and the synthesis structure. Changes to the interface cause the synthesis to behave differently; the user is then informed of this by changes back in the interface. This feedback of information from the synthesis to the interface was discussed earlier in Section 1.8 which draws parallels with the OSI 7-layer model.

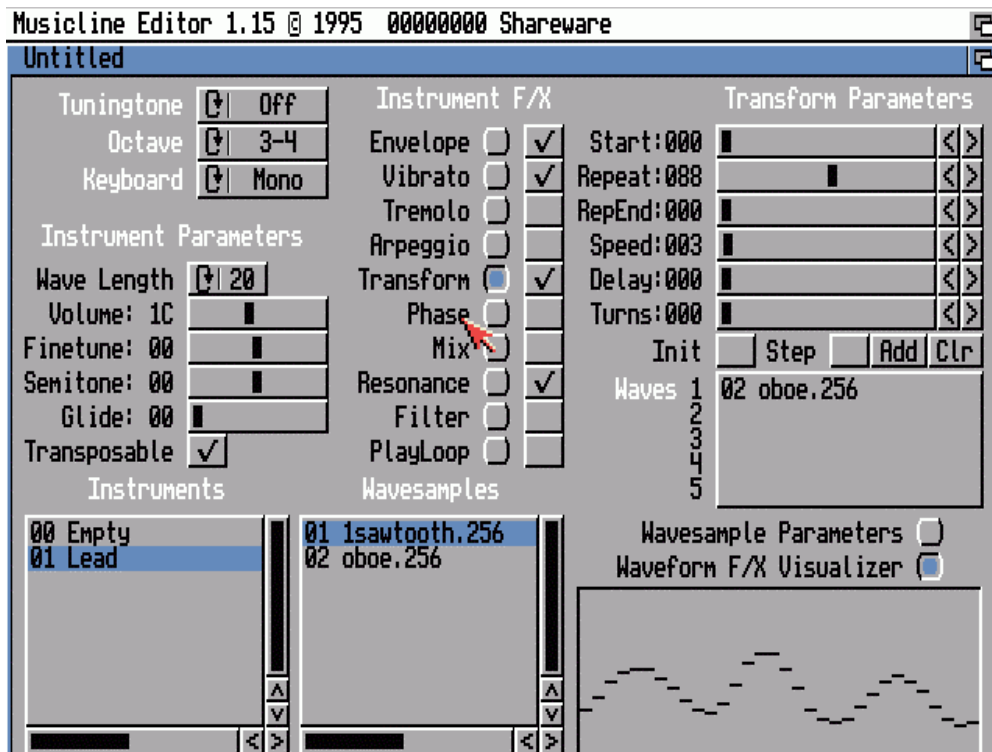


Figure 2.9 - Musicline Editor (Showing waveform F/X visualiser in lower right corner)

2.3 Modern interfaces

One of the project objectives is to provide a modern interface solution to the sound design process. Another objective is to provide a way to apply genetic algorithms to the parameter data. This section summarises the modern sources of inspiration for the interface design.

2.3.1 Genetic algorithm

Time constraints prevented the author from implementing a genetic algorithm. The initial research into genetic algorithms revealed a necessity to store and retrieve different synthesis parameters. The genetic algorithm could still be implemented at a later stage since the system was designed to be supported by a “Library storage of instruments” block in the system model in Figure 1.6 on page 12.

The inclusion of a “Library storage of instruments” block is useful, even without a genetic algorithm to connect to it since it can be used on its own. The topic of parameter storage is covered in section 2.3.3 .

The main inspiration for the genetic algorithm approach is the “Patch mutator” from the Clavia nord modular G2 editor.

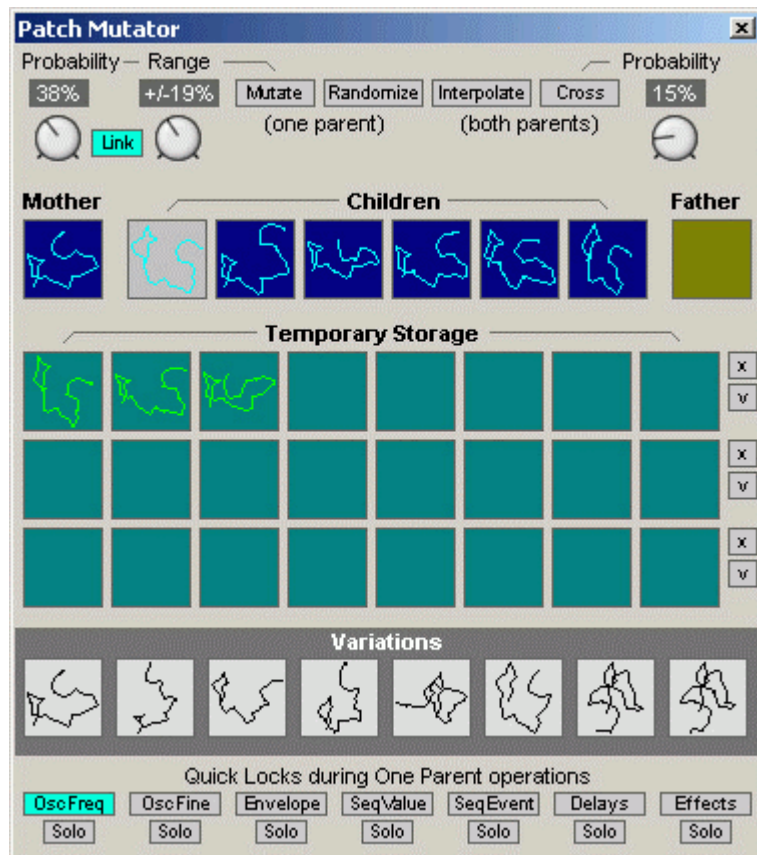


Figure 2.10 - Genetic parameter mutation
(Source: Clavia.se, 2006)

“The Patch Mutator is a toolbox for creating new patch variations guided by your ear. It will help you explore different knob settings within a certain patch by simply listening and selecting the sounds you like. Most patches can generate a vast range of different sounds, but it is tedious and difficult to explore them manually, because of the sheer number of parameters in a patch, and because of the difficulty to predict the sonic results” (Clavia.se, 2006)

2.3.2 Possible solution to “tracker instrument numbers” problem

As mentioned elsewhere (Section 1.1, 1.7, and 3.3, among others), one of the benefits of using a tracker to compose music on a system with limited polyphony is that the instrument identifiers are listed next to note triggers making it easy to conceal the limited polyphony by composing music with a high number of “instrument swaps”. One of the characteristics of chiptunes is this rapid instrument swapping. However, using a MIDI sequencer to compose in this way presents a problem since most MIDI sequencers do not provide an easy way to synchronise instrument swaps with note triggers.

The General MIDI specification uses a percussion key map whereby “On MIDI Channel 10, each MIDI Note number (or Key number) corresponds to a different drum sound” (Midi.org, 2007). Different note pitches do not trigger a single timbre at different pitches, but rather different instrument sounds at pre-specified pitches. “This is more practical than having each percussive sound allocated to its own MIDI channel which would often leave too few channels

available (for other sounds)” (Penfold, 1995) In this way, a unique MIDI trigger is assigned to a different drum kit or percussion sound. This behaviour could be a solution to the problem of MIDI sequencers not displaying instruments next to note triggers. When using the interface of the project, the user could specify which instruments to play on which notes in advance by using the interface of the project, rather than for each note by using the interface of the sequencer. This approach is most useful for situations where large numbers of different sounds are to be played, but where the pitch of these sounds is not going to change.

“Each sample in a multiple voice system can be split (using a process known as mapping) across a performance keyboard. In this way a sound can be mapped to a specific zone or range of notes so that various keys will trigger two or more different voices” (Huber, 1999) The range of 127 MIDI note pitches is split into “zones” that trigger different instrument sounds depending on which zone any pressed key lies in. For example, one sound could play when notes 0 to note 59 are pressed, and another sound could play when notes 60 to 127 are pressed.

Whilst this is mentioned under the heading of the “interface” section; implementing these techniques will have an impact on other areas as well as the user interface. There must be a sub-system which detects the note being pressed, decides which “instrument” to use, and implement a way to configure the synthesis structure appropriately before playing the note.

2.3.3 Parameter storage

An example of a desirable modern interface for arranging sounds is the commercial plugin “FM7” (Native Instruments, unknown year). This is a recreation of the Yamaha DX7 which collects synthesis parameters into a library or sound bank as shown in Figure 2.11.



Figure 2.11 - FM7 library (Arbiter.co.uk, 2007)

Compare this to a screenshot of MaxYMiser (Figure 2.4 on page 30) which shows an area in the top right allowing the user to name the defined instrument slots. The tracker approach to sound cataloguing is to save individual instrument definitions to disk as separate files. The user is responsible for cataloguing the files into groups by using the file system. This approach makes it easy to load new instrument definitions into the tracker slots to try out different orchestrations of the same music.

The FM7's method of parameter storage stems from the fact that the FM7 is a software synthesiser designed to mimic the behaviour and design of a popular hardware synthesiser - the Yamaha DX7. The FM7 has an internal storage area that can hold a number of "patches" or instrument definitions. The user can load and save entire banks of sounds with organisation of the individual sounds taking place within the sound bank. The sound bank method is compatible with the MIDI concept of "program changes" and so fits neatly with the idea of using trigger zones outlined in 2.3.2 to switch between sounds that have been defined in the bank.

These differences are highlighted in order to show that a decision must be made regarding the storage of parameter data. One solution would be to mimic the tracker method (individual instrument parameters stored on disk). If however the MIDI zone method outlined in 2.3.2 is to be implemented; complications may arise. Either the sound parameters must be recalled from disk every time an instrument is to be swapped due to a MIDI zone change, or an "internal sound bank" method must be chosen over storing individual parameters on disk.

Chapter 3 Synthesis considerations

Initial research was conducted into documentation available for the synthesis techniques surrounding chiptunes. The results were pleasing; plenty of detailed documentation exists about the hardware, a large number of datasheets and scans of original circuit diagrams and chip masks were found on the Internet, for full details see the appendices. In addition to this information, conversations were held with various developers and composers of chiptune music via email and across various IRC (Internet Relay Chat) networks. In particular, conversations with Gareth Morris (AKA gwEm) were fundamental to the author's understanding of certain synthesis techniques popular on the Atari.

Coming back to the ideas discussed in sections 1.7 and 1.8; which introduced the idea of layered systems and touched on the fact that tracker software includes instructions that form an integral part of synthesis. This is to say that the hardware chips are just one part (or layer) in the overall system of a chiptune synthesiser, and software is another layer.

As mentioned previously in section 1.8, chiptune synthesis existed in the original context as part of a complex structure (a microprocessor controlled home computer system) with many layers of functionality. The composition and playback of a chiptune involved the use of each layer of this microcomputer to some degree or other. When attempting to achieve the goal of a modern implementation of chiptune synthesis it would be a blunt approach indeed to reconstruct each layer of functionality of a microcomputer system simply because each layer is used in chiptune synthesis.

A better approach (the one undertaken for this project) is this:

- Identify each layer (see section 1.8)
- Attempt to break down the system into smaller components
- Classify the components (hardware/software/synthesis/interface, etc.)
- Focus on components that contribute to synthesis (attempt to exclude as many elements that result from the synthesis structure being part of a microcomputer)

This may involve:

- Examining hardware and software separately
- Identifying how they are linked
- Simplifying the functionality of the required components (again, excluding elements that relate to the microcomputer structure)

The reader will appreciate the following example, which emphasises the benefits of classifying user interface and synthesis techniques:

The physical hardware implementation of a chip may use discretely configurable oscillators for each wave shape. A voiced combination of “square and triangle” would therefore require two oscillators, one to play square and the other triangle. The user designs the sound “square+triangle” as a single entity, but the hardware chip would be sent separate instructions by the CPU, causing the triangle and square to sound in unison. In this way, composition software examined is guilty of abstracting the user from the hardware structure and instead implements a single “instrument”, “voice”, or “patch” as having the entire range of oscillators available on the chip. The interface of the software can obscure the true synthesis structure, resulting in a perceived synthesis structure may be markedly different to the “truth”.

This is a side effect, touched on in section 1.8, of high level interfaces being used to specify the behaviour of lower levels. Drawing parallels to a layered system: software layers determine how much control a user sitting at the highest level is given over the lower levels.

As outlined by the bullet points above: Simply looking at the software, to determine the parameters available for user configuration, is not enough. Detailed research in the hardware synthesis section is necessary in order to separate the perceived synthesis structure from the truth of the hardware capabilities. This separation helps in the extraction of the core ideas driving the synthesis structures, enabling the author to construct a system that resembles past systems, without overcomplicating the implementation.

An important fact that emerged from the synthesis research was that not all sound chips with similar features are implemented the same way. This is important when considering the conversion to a software implementation, for example: Triangle wave oscillators on one chip could be relaxational (logic gate, capacitor & resistor) on one chip, wavetable based on another, and up/down counter based on another. The first impression is to look at these three techniques and say “A triangle wave oscillator produces the same triangle waveshape, what difference does it make how it produces this shape?”, however; it depends on how much detail you look for. A chip using wavetable oscillators might include features to jump around in the wavetable, looping sections, to produce a triangle wave by default, but unusual and flexible results that a relaxational oscillator would not be able achieve.

Not everything should be examined in this much detail, it is important not to forget the broader picture: An oscillator is just a tool in a toolbox full of many others that is used to create chiptune style music. The goal here is to identify the features that are important enough to include, and those that can be discounted.

The bullet points above outlined a method whereby the hardware and software elements of synthesis are examined separately. The reason for adopting this approach is that the software driving the hardware synthesis chip often affects the diversity of the soundset produced. An example might be (as mentioned above) a hardware wavetable oscillator that allows the read pointer to jump around in the wavetable, however; in order to do this the hardware must be configured by some external controlling software.

By drawing parallels to a layered system once more: A user wishing to control a low level system can interact with a high level of the system and benefit from the abstraction afforded by the intervening layers. The hierarchy of interacting components makes the user's work simpler. In this context: the lowest level system is the hardware chip, and a number of the connecting layers involve software structures.

Given that the sound chips in question are hardware sound generation devices, it would be a reasonable first impression to assume that the (waveform, spectral characteristic, etc) output produced by a single type of chip will be the same regardless of what software is used to configure the chip. For very simple sounds this assumption is correct, but the reader should appreciate that there may be a significant difference in sound design possibilities between different pieces of software that could result in different outputs when using the same chip, but different software.

Simply expressed: "Machine A + Chip A" might produce a different output when compared with "Machine B + Chip B". The problem occurs when "Software 1" has the possibility of outputting a certain waveform (on "Machine A + Chip A"), but this possibility is not included in "Software 2". Here the synthesis structure is not confined to the hardware within the chip, but is expanded to encompass a combination of the chip and the CPU working together. The synthesis structure incorporates the CPU, chip and certain rules defined in software. Figure 3.1 explains this concept diagrammatically.

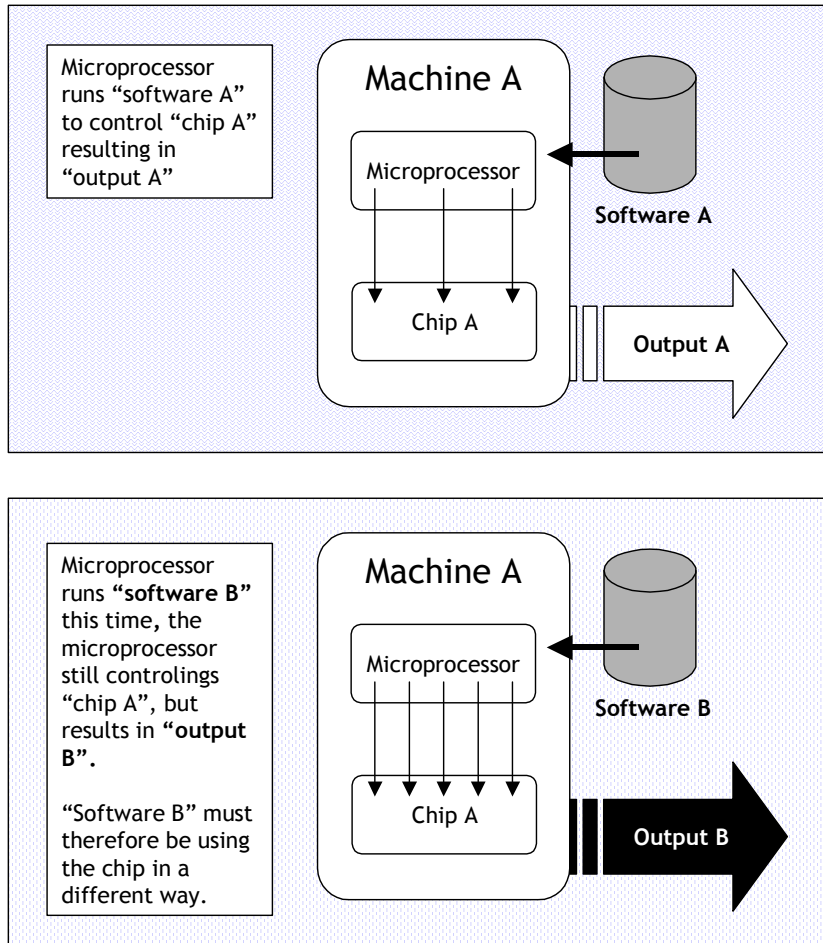


Figure 3.1 - Differences in synthesis structure

Again, referring to section 1.8 and layered systems: several layers of functionality are combined to form the overall synthesis structure. It should be noted that changes to *any* layer will result in a different output, Figure 3.1 concentrates on the implications of a change in the software layer.

To summarise these thoughts:

- The sound chips were not a self-contained synthesis structure. They could not produce “chiptune music” without the involvement of an external microprocessor
- The chiptune composition software may have included features that allowed the composer to use the sound chip and the microprocessor in unusual combinations
- These combinations may produce characteristic chiptune qualities that would be a serious omission in a project that claims to allow a user to create chiptunes
- The internal structure of chips deserve close analysis
- Software techniques deserve equally close analysis
- Breaking down the overall idea into components will promote an efficient implementation that excludes elements that result from the original synthesis structures being part of a microcomputer
- Certain features are more common than others
- Only by examining a wide variety of chips and software can commonality be discerned

In short: when designing the synthesis structure the author will draw from historic chip hardware designs AND historic software designs.

3.1 Historic hardware synthesis

In Section 1.4, one of the initial objectives was to provide a “complete set of sound generation and design techniques that cover the entire range of timbres produced by the custom chips for which chiptunes were composed”. The research into the internal designs of different sound chip hardware was started with the intention of fulfilling this objective. However, as the list of techniques became larger, it was obvious that it would not be possible to create a software version of every hardware chip and every software technique.

The research is presented as tables containing detailed information about each chip. To prevent such a large number of tables breaking up the flow of the report, they have been placed in an appendix. Table 3.1 summarises the sound hardware that was examined during the research:

Computer system	CPU	Sound chip
Atari (ST)	Motorola 680x0	Yamaha YM2149 (identical pinout to AY8190/8912)
Atari 2600	MOS 6507	Custom TIA (Television Interface Adapter)
Commodore 64	Motorola	Custom SID 6581 & 8580 (Sound Interface Device)
Commodore Amiga	Motorola 680x0	Custom "Paula"
MSX	Zilog Z80	General Instruments AY (8190, 8912) Optional Konami PSG Y8950 Optional Yamaha YM2413
Nintendo Entertainment System	MOS 6502	Cy2A03 and DMC (Delta Modulation Controller)
Nintendo Gameboy	Modified Zilog Z80	Custom (feature list extremely similar to 2A03)
ZX Spectrum (16k, 48k)	Zilog Z80	NONE (The CPU directly produced 1-bit sound, fed to a simple onboard amplifier)
ZX Spectrum (128, +2, +3)	Zilog Z80	General Instruments AY (8190, 8912)
Acorn BBC, Tomy Tutor	MOS 6502	Texas Instruments SN67489AN

Table 3.1- Summary of sound hardware examined during research

This list was compiled from various sources, described in full in the appendices.

A great deal of effort was applied in analysing and making sure the author firmly understood the information presented in the datasheets for the hardware. The author examined many circuit diagrams and chip masks for various chips, a great deal of time was spent learning and absorbing the underlying principles.

The placement of this information in an appendix should not be interpreted as an indication that it is less important than the main text; it is simply placed there to avoid breaking the flow of the text. The reader is strongly encouraged to read this information, and to evaluate it as part of the report.

3.2 Historic software synthesis

Most of the chips encountered (See appendices) were essentially hardware oscillators, designed to be used in conjunction with a microprocessor. The chip creates audio signals only after being configured by the CPU. These chips with internal oscillators can be thought of as "state machines", with internal structures that can be configured in a number of different ways and remain in that state until a new state is specified.

Early music composed for these kinds of chips kept the CPU's involvement to a minimum. The CPU would configure the state of the sound chip by setting the desired hardware synthesis parameters for each "note" in a musical piece, the CPU would keep track of what notes to

play, how long each note lasts, and would configure the new synthesis parameters in the gaps between note triggers.

As time progressed, various techniques emerged that aimed to extend the sound palette of these chips, subverting the limited hardware by exploiting various side effects.

MaxYMiser (Morris, 2006) is an example of a very modern piece of software that encapsulates a great many software augmented synthesis techniques for the YM2419. It collects together in one package an extremely wide range of techniques developed by many other people that have been used elsewhere in other pieces of software for the Atari such as MagicSynth (Seeman, 1997), and MusicMon (Lautenbach, 2006).

The YM2149 used in the Atari is almost a direct clone of the General Instruments AY-3-8910 and has envelope generators that can be set to loop, the original intention is that these be used to produce tremolo (amplitude variations) for the square oscillators. The looping envelope waveshape is selectable between triangle and saw.

Through email conversations (Morris, 2007) with the author of MaxYMiser; a brief summary of popular ATARI chiptune synthesis was constructed and is presented below:

1. The YM2149 features a software controlled attenuator for each square wave channel. By using CPU interrupts to adjust the attenuation level at audio rates, amplitude or ring-modulation of the square wave carrier oscillator is possible. The waveshape of the modulation “oscillator” need not also be square, since the sequence of attenuation levels can take any values. In Atari chiptune composer terminology, this is known as “Sid Voice” due to the similarities of the Commodore 64 SID chip with an intentional design feature of amplitude or ring-modulation between oscillators of various waveshapes.

When the frequency of a square wave oscillator is in the audio frequency range, then the envelope generator will be performing amplitude or ring-modulation on the square wave oscillator. This produces a large number of sidebands due to the harmonic content of the square carrier wave. Subtle changes in the square carrier frequency produce very interesting aural results.

2. If the modulation “oscillator” attenuator sequence described in the above method acts on an oscillator which is “paused”. The output of the carrier oscillator has a DC value only, the attenuator values are the only oscillations at the output. This technique is used to produce sinewaves on a chip which has no sinewave oscillator, and can even be used to replay PCM samples.

Most techniques researched for other chips are minor variations of the themes discussed here, using hardware interrupts at audio frequencies to alter the state of the custom sound chip in a way that produces an entirely new waveform.

3. The envelope generators can be configured to oscillate at audio rates by using hardware interrupts clocked at audio frequencies. If the envelope acts on an oscillator which is “paused” (i.e. the output is a DC value) the looping envelope waveform is perceived as an oscillating tone. In Atari chiptune composer terminology this is known as a “Buzzer” effect.

4. A side effect of the envelope generator design on the YM2149 is that, when in looping mode, the phase of the envelope is reset to zero after the register that sets the envelope shape is written to. If the envelope generator is configured in “Buzzer” mode (i.e. looping envelope oscillations at audio frequencies), when the phase of the envelope generator is reset, an audible click will result if the phase is not reset at a zero crossing of the envelope generators output. By using a further interrupt timer, also at audio rates, to reset the phase of the envelope generator; an extremely interesting audio effect is produced called “Sync Buzzer” in Atari chiptune composer terminology.

Referring back to the idea of layered systems introduced in Section 1.8, these techniques use the CPU layer to control elements of the sound hardware layers in a non standard way. These examples are intended to demonstrate techniques involving the co-operation of hardware and software synthesis techniques.

The Sinclair 48k ZX spectrum is an excellent platform to demonstrate the principle of software extending the capabilities of sound hardware taken to its logical extreme. The sound hardware present in a 48k (and 16k) ZX Spectrum was possibly the simplest imaginable. It consisted of an internal piezoelectric loudspeaker connected to a single digital output through a small transistor amplifier. It was designed to be used with a ROM routine that produced 1-bit monophonic square wave “beeps” by toggling the digital output. For more information, including a circuit diagram, see the appendices.

Most compositions for games released on the 48k spectrum are of this single channel, monophonic square wave “beep” variety. There are however; some very exceptional compositions that utilise custom sound routines that use the Z80 CPU directly to toggle the 1-bit output.

Tim Follin was a game music programmer for the ZX Spectrum between 1985 and 1987. He “programmed a single channel phase-shift routine, a three channel routine, and a five channel routine” (Follin, 2007). The five channel routine is described by Tim Follin himself in an interview in a ZX spectrum magazine, and in more detail in a post on an internet newsgroup below.

“It worked by using five of the Z80’s operators (C, D, E, H, L) in a loop, which were all counting down to 0. When they hit 0, I’d make a click at a certain volume (governed by the delay between switching the speaker on and off) and reset the operator to it’s original value. Every so many cycles of the loop, I’d jump out to change notes and things. To keep the speed up, it used self-modifying code, which meant that the main loop could just have simple commands like ‘set H to 110’, while the code outside the main loop changed the code inside the main loop by writing over it with the new values. The only draw back was that it sounded like a vacuum cleaner with nails stuck in it.” (Comps.sys.sinclair newsgroup, 2000b)

“The program consisted of a tight loop, set to play for about 1/30th of a second, before breaking out to alter the parameters within the loop (writing directly over the code within the loop, to speed it all up). In the break, it also checked for things such as the break key, and played the drum ‘samples’. In the loop, I had three - or as many channels as it played counters (registers b, c, d, e, etc) counting down continuously until they reached zero, at which point it would switch the speaker on and off; the delay between the on and off controled the volume. This obviously meant that the

delay between clicks had to be balanced by another delay loop, to keep the overall time the same, which obviously slowed the whole loop down quite a bit. But there you go! It worked... Sort of.”

(Comp.sys.sinclair newsgroup, 2000a)

Another quite different method is that found in the game “SAVAGE”, with music (and sound playback routines) by Jason. C. Brooke. This technique involves the DAC hardware driving an internal speaker with 1-bit information at an extremely high frequency above the audio range. Using this method it is possible to create an approximation of a desired (non-square) wave by driving the DAC with squarewaves that exceed the audio bandwidth causing the piezoelectric loudspeaker plate to move around at audio frequencies.

This method of over sampling was also used on the IBM PC: “magic mushroom demo”. For more information see the “history of soundcards” (Crossfire-designs.de, 2007), SPEAKER.txt and PIT.txt (FELDMAN, M., 2007a, and 2007b)

3.2.1 Software with similar goals to the project

During the research into software techniques for chiptune synthesis, software written to emulate the sound of older machines was discovered. Some of this software includes interesting features that extend what even the systems they emulate could do by hybridising features of the emulated and the emulating system.

For example: Abyss’HigherExperience (AHX) and MusicLineTracker (MLine) both use the Commodore Amiga’s CPU to generate audio waveforms that resemble those produced by the Commodore 64’s SID chip. Where the SID has only 4 channels, AHX and MLine allow the user to compose with many more channels, since it is the CPU generating the waveforms and not a finite collection of hardware.

Both AHX and Mline allow the composer to produce music in the chiptune genre on a machine that, by design, has sound hardware is supposed to surpass the limited sounding chiptune timbres. A curious aspect of this intentional construction of a limited sound palette is that both AHX and MLine allow the gradual lifting of restrictions, allowing the composer to blend chiptune timbres (in the emulation of the C64 SID chip) with more modern sounds produced by the Amiga’s hardware.

Octamed is another example of Amiga software that allows this blending of old and new sounds. It has a SynthSound editor that allows scripting of CPU generated waveforms based on wavetables. The SynthSound editor can be used to create timbres similar to the C64.

FamiTracker (Shoodot.net, 2007), and VortexTracker (Bulba, 2000) are both pieces of modern composition software that emulate the sound hardware of other computer systems. The accuracy of the emulation of sound hardware is a high priority for both of these systems. FamiTracker emulates the 2A03 chip from the Nintendo Entertainment System. VortexTracker emulates the AY-3-8910 and YM2149 chips from the ZX spectrum and Atari. These systems take the form of a tracker interface connected to the sound emulation routines. Users who compose music with these pieces of software are most likely to be concerned with an accurate recreation of a sound design technique used on these systems and comfortable with the concepts (and restrictions) a tracker presents.

Goat tracker (Homeftp.net, 2007) takes the idea of modern composition software with sound generation accuracy extremely seriously by supporting PCI cards that feature timing hardware and circuits duplicated from the Commodore 64 motherboard, and a sockets for a real SID chip. For more information see the CatWeasel (Individual computers, 2007) and HardSID (Hard Software, 2003) websites.

AYEMUL, JAM (Creamhq.de, 2007), and DELIPLAYER (Delitracker.com, 2007) are all examples of modern software designed just to replay existing chiptunes from older systems. They take the idea of very accurate replay code exhibited by FamiTracker and VortexTracker but do not present an interface to compose music. The purpose of these systems is to enjoy music composed for these systems in the past, without the original hardware. Some players support the CatWeasel and HardSID devices for those who do own original hardware.

T'SoundSystem (Toyoshima, 2004) is a piece of software from a Japanese author that aims to simulate some characteristic sound generation techniques found in past games consoles such as simple wavetable oscillators, noise generators, FM timbres, etc. It has no composition interface; the user must use a text editor and markup language to compose music. New users are likely to find this method of composition extremely difficult.

QuadraSID (Refx.net, 2007), and YMVst (Morris, 2006) are both modern VST plugins that emulate the characteristics of a specific sound chip of the past. QuadraSID emulates four instances of the C64 SID chip (16 oscillators in total), whilst YMVst emulates a single channel of the Atari YM2149, and requires 4 instances to be loaded to emulate an entire chip. These pieces of software are designed to integrate with a modern MIDI/Audio sequencer that supports VST plugins. The ability to be MIDI controlled also enables them to be used in a live situation, users of FamiTracker and VortexTracker are unable to audition melodies without first programming them into the tracker interface whereas users of the VST plugin systems are able to play ideas directly on a keyboard.

Mmonoplayer "buza" (Mmonoplayer.com, 2007) is a collection of MaxMSP externals that emulate, among others, the sound of the 2A03 and the Atari2600 TIA hardware.

Little Sound DJ (Littlesounddj.com, 2007) and Nanoloop (Wittchow, 2007) are both examples of modern composition software for the Nintendo Gameboy. Little Sound DJ users can build a Gameboy to MIDI cable that allows synchronisation with other MIDI devices.

3.3 Modern techniques

As mentioned in Section 1.1, 1.7, and 2.3.2 , a particular characteristic of chiptunes (used for concealing the often limited polyphony of the sound chips) was to voice chords as very fast arpeggios, breaking each note in the chord up into individual notes and playing them as quickly as one note every 25-30 milliseconds. Most MIDI sequencers do not provide an easy way to break a chord into arpeggiations with notes being triggered this quickly.

Quadrasiid (Refx.net, 2007) was mentioned in section 3.2.1 , it applies arpeggiation to a sound by converting any two or more simultaneously held down MIDI notes into a sequence of individual monophonic note triggers with a specified delay between each note. Monophonic input is left untouched. In this way, a pre-recorded MIDI file for a polyphonic synthesiser is interpreted flawlessly; single notes are played as normal, and chords are replaced by arpeggiations when necessary. This technique is also useful for live performance, since a monophonic sound can be suddenly arpeggiated to add expression to any performance.

Mda JX10 (Mda-vst.com, 2007) features a legato portamento that behaves in a similar manner to the QuadraSID arpeggiator. It causes two or more simultaneously held notes to glide to the most recently received pitch, but passes through single notes untouched. In a live situation, a user can perform a melody that automatically changes between abrupt staccato notes with defined pitch changes, and lyrical phrases with long notes that microtonally slide between pitches.

The reader should appreciate that implementing a solution to these problems will have an impact on the synthesis structure, and on the user interface. The solutions related to each area are discussed in the implementation chapter.

Chapter 4 Summary of findings

4.1 Major decisions – changes to objectives

Due to time and resource restrictions, it was not possible to use all the information gathered during the research to create a software version of every hardware chip and every software technique. This section returns briefly to the topic of project management, describing the decisions made in order to reduce the scope of the implemented techniques, and highlights the reasons for choosing to implement a particular set of synthesis components.

The original objective was to recreate sound characteristics of a small collection of specific chips, creating a system that combines many “classic” synthesis techniques; it is no longer an objective of this project to recreate such a wide variety of techniques specific to different systems and to combine them into one synthesiser.

It is clearly too large a task to design and implement in software, for this project, the entire catalogue of sound chips popular in home computers. Some reduction in scope is necessary.

The chosen method for reducing the scope was to examine the datasheets and design documentation for the historic chips (see appendices) and to look for common behaviours or design features. This approach was taken in order to satisfy the new objective of creating a generic chiptune sound synthesis structure with common synthesis techniques from different chips.

The implications of this decision to reduce the complexity of the project objectives, involve the following actions:

- Examine the documentation for the different chips
- Identify the common factors of design / synthesis / implementation
- Identify software features that contribute to the sound
- Produce specification for the various synthesis components that were identified
- Implement the synthesis structure in software.

It is worth pointing out at this stage that a change to the project objectives, in terms of the synthesis structure, affected other areas of the project very little. This is due to the decision (outlined in section 1.5.1) to keep the synthesis structure development separate from the user interface.

4.2 Summary of findings

This section summarises the findings related to both hardware and software synthesis. It forms the basis for a list of requirements that the project must meet.

Firstly, referring to the hardware examined: each chip features a simple synthesis structure with a number of oscillators, level attenuators, and a mixer. Almost every chip has limited polyphony, (i.e. a finite number of oscillators are available simultaneously). Most datasheets for the chips group synthesis parameters for each oscillator block into “hardware channels”. Parallels can be drawn with scoring a musical piece for 4-part harmony (similar to a “barbershop” quartet). The individual scores for each “voice” are programmed by the CPU, with values being poked into the appropriate “hardware channel” on the chip.

Almost every chip seemed to have some sort of random number generator that can be used to generate a sequence at a high enough rate to be heard as “white noise”. Some can generate “pitched noise” where a short sequence of noise is repeated.

Almost every hardware chip examined also included a square wave generator, and most chips could alter the duty cycle of this square wave. Some chips could produce hardware waveforms other than square/pulse, the most common alternative waveform being a triangle.

Only one or two exceptional chips (the SID and AY/YM clones in particular) have unusual internal synthesis structures that allow amplitude or ring-modulation and oscillator phase synchronisation. However, these advanced synthesis techniques simply augment a simple underlying synthesis structure of oscillators, level attenuators, and a mixer.

Not all chips have the same range across every hardware oscillator. The 2A03 has an oscillator with a range best suited to playing musical phrases with low frequencies and another oscillator with a range best suited to phrases with higher frequencies.

Every chip examined had some sort of internal level attenuator for each hardware oscillator. Most of the chips examined had additional envelope generators that could generate appropriate sequences of level attenuation, although some chips require the use of the CPU to program the level attenuators whilst a note is being played to produce envelope-like effects.

Each and every chip examined resembled a “state machine”. After the parameters (such as oscillator frequency, envelope shape, and overall level) are configured, the chip freewheels, remaining in this state and producing an audio output continuously until the CPU configures a new state.

Directing the focus now to software techniques used to augment the hardware: but still with reference to the idea of “state machines”: most of these techniques involved creating complex sequences of parameter changes, instructing the CPU to program the hardware to revert to different states in a rhythmic manner. These techniques involve programming the sound chip to change state *during* notes, changing the character of the sound *whilst a note is being played*.

Almost all other software techniques used the chip in a “non-standard” way (i.e. using features that are not documented in the hardware datasheets, using components to perform functions they were not designed to do; envelope generators as oscillators, mixers as DACs (Digital to Analogue Converters), updating parameters at audio rates, etc) These methods were developed to extend to the sound palette available to composers, allowing them to compose with CPU generated waveforms instead of hardware oscillators. In this way, a chip could produce many more sounds than those documented in the hardware specifications.

4.3 Specifications and requirements

The research into the historic chips showed that the most common type of waveform is the square wave (often with a variable duty-cycle); the next most common is pseudo-random noise (commonly selectable between periodic and non-periodic noise). The next most common waveform is a triangle wave. Based on their common occurrence in the chips examined, these waveforms should be implemented in the project.

All of the examined chips had monophonic oscillators and grouped each oscillator block into discrete channels; for authenticity, each oscillator block in the project should be monophonic. This modular approach means a single synthesiser structure can be developed once, then duplicated for the required number of channels. For simplicity and due to the fact that many chips featured just four note polyphony; the number of channels implemented in the project should initially be limited to four.

Some of the chips examined had stereo outputs and mixed their individual hardware channels together at the output by grouping channels to the left or right output only. The author suggests that a finer level of control over the stereo position of each hardware channel need not be a hindrance to compositions aiming to be authentic to “hard left or right panned” compositions, since the pan positions can still be set this way if desired

Since all the chips examined were programmed as “state machines”, this behaviour should be recreated in the project in some way. This will enable the user to specify sequences (or patterns) of synthesis parameter values that will be stepped through at a certain speed when a note trigger is received.

Several features such as monophonic arpeggiation of polyphonic midi input, bending of notes when polyphonic midi input is received, midi drum maps zone splitting were all mentioned and their inclusion in the project justified in section 3.3.

Chapter 5 Implementation

This chapter briefly re-iterates the design concept behind the entire system and how each component fits together. The approach is to examine the entire structure in gradually increasing detail. It includes explanations of commonly used MaxMSP objects and concepts to help readers who are unfamiliar with the environment understand better.

Detailed notes of the implementation for each module are included, outlining the requirements and purpose of each module, any options problems or decisions that result in changes to the initial design as a result of prototyping. The final solution of each module is explained by referring to the final MaxMSP structure, and any potential future developments

are identified. Test data, results, and recommended bounds for inputs and outputs for each module are included to aid software maintenance tasks. A prototype was constructed in PureData as a personal project before the final year started; a screenshot is shown in Figure 5.1 below.

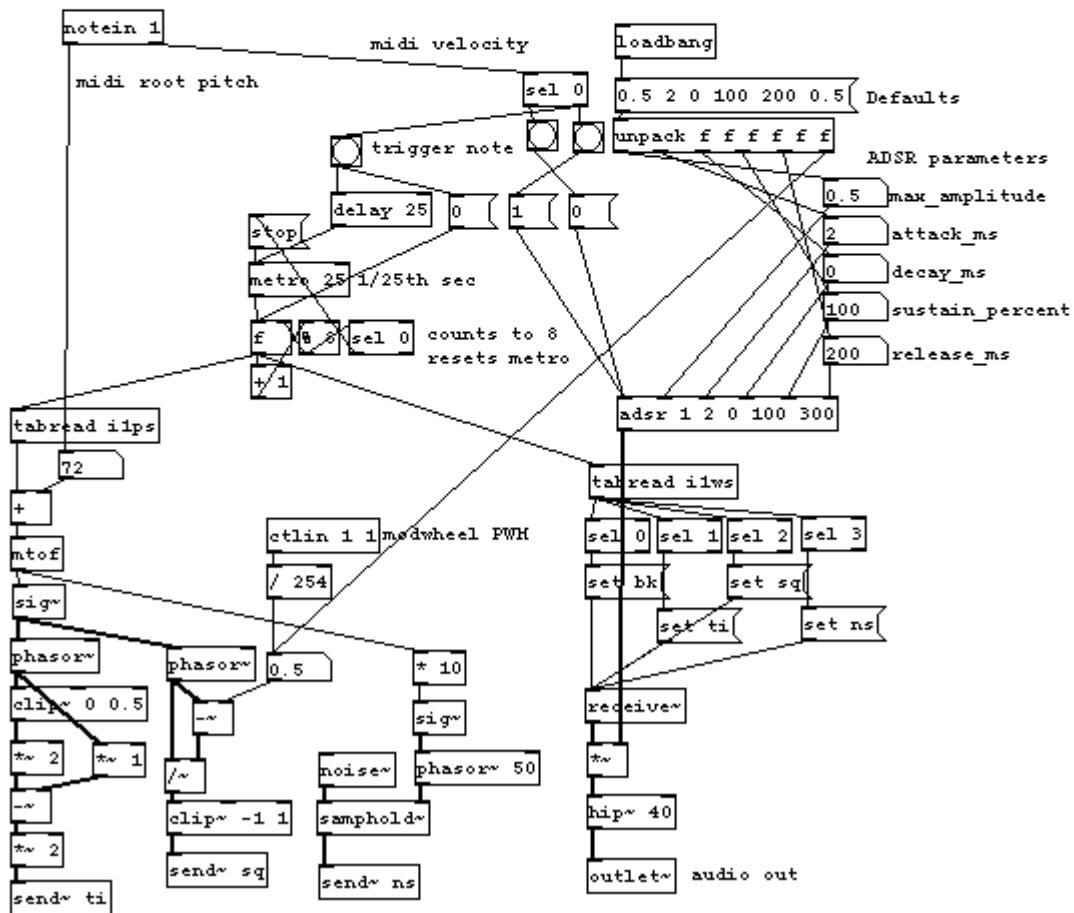
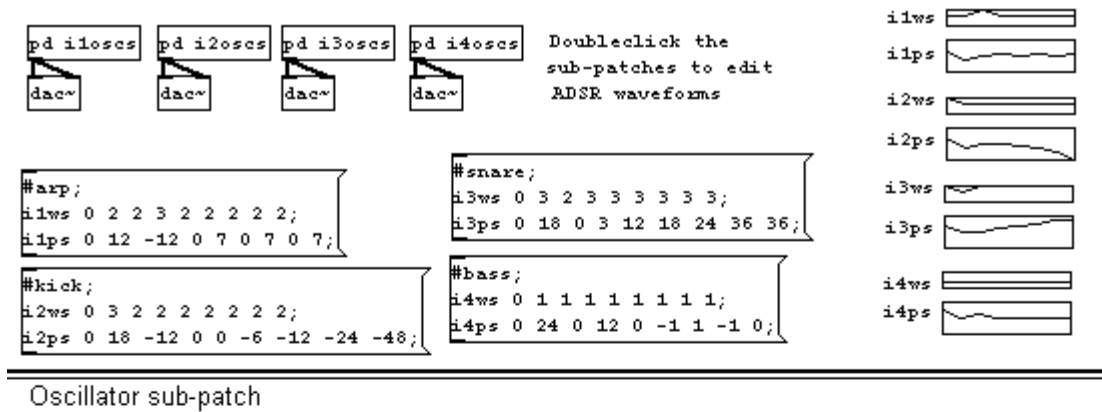


Figure 5.1 - PureData prototype structure (one oscillator shown)

In the early stages of the project, the author converted the functionality of the prototype to an implementation in MaxMSP with an attempt at embellishing the crude PureData prototype interface (which requires the user to type correctly formatted sequences). The author experimented with the native interface objects in MaxMSP and with the efficiency of generating control sequences and wave shaping.

A large number of MaxMSP “patches” (files containing the structure of a group of connected objects) evolved into a pool of data that could be treated as larger components to be used to form the final structure.

5.1 MaxMSP terms and conventions

In order to properly explain the final solution, the following sections use MaxMSP terms and conventions quite freely, which may present a problem if the reader is unfamiliar with these. This section aims to make the reader more comfortable in their understanding of MaxMSP to aid the reading of the following sections.

5.1.1 Fundamental concepts and terms

Various fundamental concepts and terms commonly used throughout the project are summarised below.

Max and MSP as separate concepts

Early versions of MaxMSP did not include the “MSP” part of the name. A very broad explanation for this is that “Max” was essentially a control-rate processing environment for MIDI data. Signal-rate processing was added at a later stage.

The term MSP stands for Max Signal Processing, and is an extension to the basic “Max” platform to allow signal-rate processing of audio data. The names of objects designed for MSP processing usually end in the tilde “~” character to indicate that the object is an MSP object and not a standard Max object. Signal rate messages are processed at the audio sample rate of the system and MUST produce values every sample (at the sample rate) whereas control-rate objects need only produce a new message when needed to (for example, when a value changes).

Object

This term is difficult to describe concisely: A broad answer would be “an object represents a block of functionality, a package of underlying instructions for the computer to perform”, however; the amount of processing performed by an object varies. Primitive objects, Sub-patches, and abstractions are among the many things that can be referred to as an “object” in a MaxMSP structure. Think of an object like a function machine, in that they perform actions on data fed to them.

Inlets and Outlets

Each object can have one or more inlets and outlets. These allow data to flow into and out of the object. Certain objects allow the nature of the processing performed by an object to be altered by sending configuration messages to certain inlets.

Argument

Another way to configure the internal processing structure of an object is to pass it arguments. This is very similar to the idea of passing an argument to a function in a high-level language such as C or Pascal. Arguments can be “de-referenced” in sub-patches and abstractions by using hash-notation “#1” for the first argument, “#2” for the second, etc.

Cable

Objects can be connected together with cables. Cables represent the flow of information between objects, connecting the outlets of one object to the inlets of other objects. Max objects (control-rate objects) are connected together with thin single pixel solid-colour cables; signal-rate objects (MSP objects) are connected together with thicker 3-4 pixel striped-colour cables.

Cables versus send + receive objects

Objects in a MaxMSP environment can be connected with cables, or by using [send] and [receive] objects in pairs. A [send] object can be abbreviated to [s], likewise with [receive] and [r]. By using arguments to group an [s] object to an [r] object, the need for connecting cables is eliminated; information flows through virtual cable between the [s] and [r] pair.

Message

Again, this term is difficult to describe concisely: A message is a term used to describe the (usually control-rate) information that can flow from an object's outlet, through a cable, and into an inlet of another object. Some control-rate message types are: integer, floating-point, list, symbol, bang...etc.

Patch

A patch is a representation of a collection of objects, their connections, and their structure. The term "patch" often refers to the file stored on the disk that contains the instructions for what objects to connect, which outlets to connect to which inlets, etc. An analogy describing a patch is that "If objects are building blocks, then patches are the buildings."

Sub-patch

A "sub-patch" is simply a patch that has been embedded in another patch. By using an [inlet] object, and an [outlet] object a sub-patch can appear as if it were "just another object" with inlets, outlets, and some form of processing going on inside.

A very important point to make is that sub-patches are not self-contained: When a patch is made that involves sub-patches, the sub-patches are saved *as part of the main patch structure*, not separately like an abstraction.

Abstraction

An abstraction is similar to a "sub-patch", however the structure of an abstraction is dealt with as a separate, self-contained structure that can exist outside of the main patch. Abstractions make it possible to re-use code by creating a patch that contains functionality that is likely to be used multiple times but operating on or with different data. An abstraction enables large and complex structures to appear as a single object in a max patch, reducing graphical clutter. Abstractions are identified by the filename of the patch file containing the abstraction's structure.

External

An external is a collection of C-language code that has been compiled using the MaxMSP software development kit (SDK). Although large and complex structures can be built using MaxMSP objects, patches, sub-patches and abstractions; the attraction of externals is that of the computational efficiency afforded by programming certain

types of functionality in C. If certain functionality is desired, but the graphically programmed prototype structure is slow (or unnecessarily large for whatever reason), the idea is that the MaxMSP SDK provides a convenient way to add this functionality to MaxMSP with relative ease, potentially with an increase in performance.

5.1.2 Commonly used objects

To give the reader some idea of the types of objects used throughout the project; descriptions of some commonly used objects are paraphrased from the MaxMSP reference manual (Cycling'74, 2005). The reader is strongly encouraged to consult this manual since it explains every object available in MaxMSP (with the obvious exception of third party externals) and is available from the Cycling'74 website for free.

Numeric value storage [int], and [float]

Control-rate: Temporary storage of integer and floating point numbers as opposed to direct transmission of values between objects via cables. The [i] and [f] objects are abbreviations.

Arithmetic objects [+],[+~],[-],[~-],[*],[*~],[/] and [/~]

Control-rate and signal-rate: Provide means to scale values by multiplication, shift values by addition or subtraction, and compute ratios by division. The symbols used for identifying the operations to be performed are identical to the C language.

Logical operators [=],[=~],[!=],[!=~],[>],[>~],[<],[<~],[&&],[| |]..etc

Control-rate and signal-rate: Provide a means of constructing logical statements to control the processing of certain data depending on its form or content. The symbols used for identifying the operations to be performed are (mostly) identical to the C language.

Flow control [gate]

Control-rate: Blocks (or passes) the flow of messages depending on a certain condition.

Flow control [route]

Control-rate: Distributes list messages to different locations depending on the first element of the list matching a given value.

Timing [metro]

Control-rate: Produces a "bang" message at a specified interval in milliseconds.

Counting [counter]

Control-rate: Produces a control-rate integer output each time a "bang" is received that increases or decreases from a specified value until a specified minimum (or maximum) is reached, whereupon the count repeats.

Initialisation [loadbang]

Control-rate: Sends a "bang" message as soon as the patch is loaded. This is useful to initialise structures that need to be configured automatically in some way before the user begins to use them.

Patch display [pcontrol]

This object (among other things) facilitates the loading of patches without the user using MaxMSP's "File->Open" menu item. Patch windows can be opened by clicking buttons that send appropriate messages to the [pcontrol] object.

Patch display [bpatcher]

This object allows sub-patches to be shown in an embedded window in the calling patch. It allows sub-patches with user interface components to be displayed as if they were part of the calling patch.

Patch display [thispatcher]

This object (among other things) can be used to control the way a sub-patch is viewed. When used with the [bpatcher] object; it can be used to scroll the [bpatcher] object view to reveal different areas of the sub-patch.

It can also be used to alter the Windows/MacOS window properties for MaxMSP child windows, turning off the "Minimise" and "Restore" buttons and prohibiting the user from dragging, scrolling, or changing the size of the window displaying the sub-patch.

Datatype conversion [tosymbol], and [fromsymbol]

Control-rate: Convert a long list (maximum 2048 characters) of separate values into a single entity referred to as a "symbol".

List messages [pack] and [unpack]

Control-rate: Concatenates separate control-rate numeric values into a list message. It is useful for combining pairs of related numbers into a single message that can be sent through one control-rate cable. The [unpack] object, separates the list message into individual values again.

Datatype conversion [sig~]

Conversion of a control-rate input to a continuous stream (i.e. a value for every sampling period in terms of the audio system sample rate) of signal-rate values. Integer and floating-point control-rate inputs are converted to floating point signal-rate versions.

Signal switching [selector~]

Signal-rate: Multi-position switch that will pass through an input from different signal-rate sources depending on which one is selected.

Ramp generator [line~]

Signal-rate: This object generates signals that consist of ramps, rising or falling from a starting value to an ending value over a period of time. It can be used to generate complex signals used for a variety of control purposes.

Wavetable storage [buffer~]

Signal-rate: This object is a storage space for a sequence of values that can be accessed with (among others) the [index~] object.

Wavetable playback [index~]

Signal-rate: This object reads values stored in a [buffer~]. When combined with another signal-rate object producing a sequence of index points that relate to positions in the [buffer~] it can form a wavetable lookup oscillator.

Roads (1996) describes this combination as a "table-lookup" oscillator, which repeatedly scans a wavetable in memory to generate a periodic output.

5.2 Overview of system

Referring back to the system model (Figure 1.6 on page 12) the main MaxMSP structure is assembled from smaller structures, themselves assembled from smaller structures. This report section examines each structure in turn, starting from the main patch and gradually examining the smaller structures within, revealing more detail and lower level functionality.

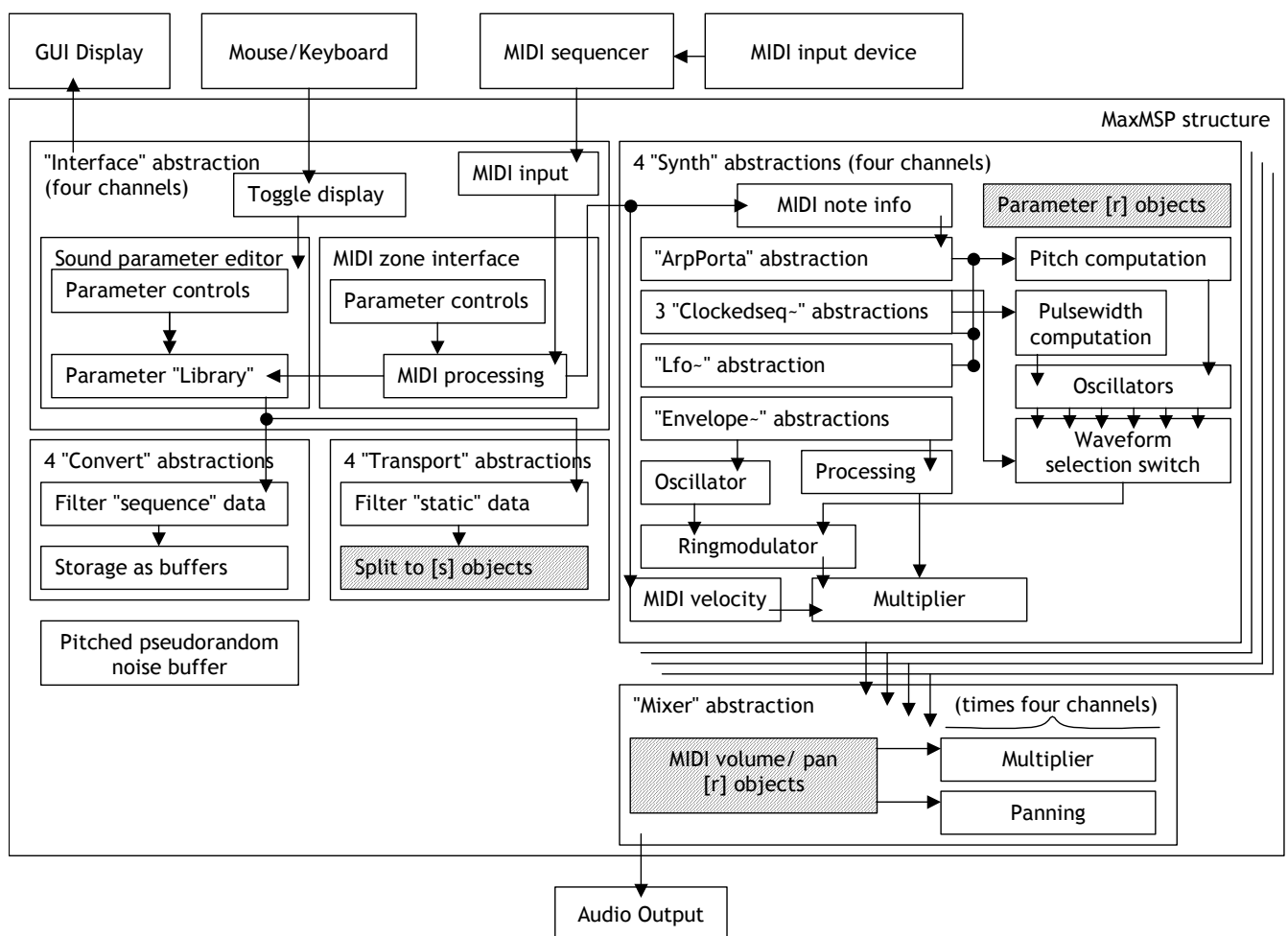


Figure 5.2 - Overview of system, closely matching MaxMSP structure

Figure 5.2 closely resembles the structure of the system within MaxMSP. Each block represents a structure that may have structures with more blocks inside. The blocks outside the "MaxMSP structure" represent external elements such as the Mouse and Keyboard, and a MIDI sequencer running in parallel to MaxMSP. The way the external MIDI elements have been arranged in Figure 5.2 represent just one possible way the system could be set up. This (a

MIDI input device connected through the MIDI sequencer) is the way the author has primarily tested the system.

The structure outlined in Figure 5.2 is quite complicated; the arrows showing the flow of data make it more so. Double-headed arrows indicate instructions to recall parameters from the “library” storage. It may help to compare it to the original system model in Figure 1.6 (page 12). Two of the components (Interface, Synthesis) can be traced back to this original system model. The genetic algorithm was not implemented, and the remaining functionality of the parameters section described in the original system model is spread over the “Parameter Library” and the “Convert” and “Transport” abstractions.

For more information, see section 5.5.1 and 5.6 which explain the use of the [convertdata] and [transportdata] abstractions in more detail. Figure 5.28 on page 97 explains the use of these abstractions graphically.

Options, Problems, Decisions

The final system was developed from the specification and requirements outlined in Section 4.3, but since there are always multiple solutions to an engineering problem; the final structure was arrived at after a considerable amount of experimentation and effort.

As explained later, many revisions of the structure were made due to minor changes in other areas. Changes in the way values are generated at the user interface cause a chain reaction that necessitates a change to the way that the values are accessed, stored, routed, and processed. Every revision required a large amount of work.

18 major revisions were made of the structure, with many more minor changes. For each revision, many tasks were carried out in parallel such as design, implementing, testing, modifying, integrating code from previous versions etc. After so much experimental development, the specification for the required components was eventually evolved and implemented.

Final solution

Figure 5.3 below shows the main MaxMSP patch. A comparison with Figure 5.2 on page 55 will show that the “interface”, “convertdata”, “transportdata”, “synth”, “mixer” and “noisebuffer” abstractions all appear at the first level inside the main MaxMSP structure. A minor exception is that (for space considerations) the “convertdata” and “transportdata” abstractions are labeled “Convert” and “Transport” in Figure 5.2.

There are four channels, as per the original system model Figure 1.6 on page 12. Each “channel” comprises of an group of four abstractions: “interface”, “convertdata”, “transportdata”, and “synth”. Again, a comparison with Figure 5.2 above will show that the MIDI control data flows between the “interface” and “synth” abstractions by means of a cable, however the synthesis parameters flow by [s] and [r] objects in the “convertdata” and “transportdata” abstractions. This will be explained in more detail later.

The group of abstractions that make up a channel are given identical numeric arguments ranging from 1 to 3. These arguments are extracted inside the abstractions and used to address the MIDI input channels, and to uniquely identify objects to prevent channels influencing another’s data due to them referencing a variable with an identical name.

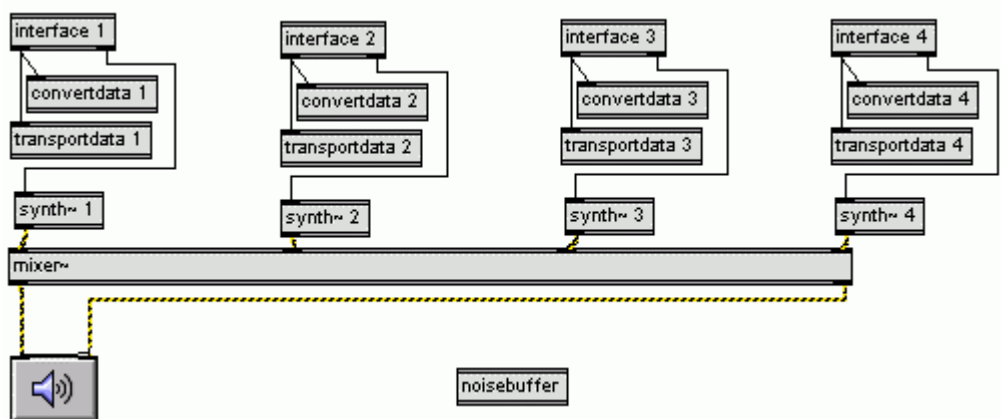


Figure 5.3 - Main MaxMSP structure

An examination of each of the abstractions shown in Figure 5.3 follows.

5.3 Synthesis

The final synthesis structure in overview is presented in Figure 5.4:

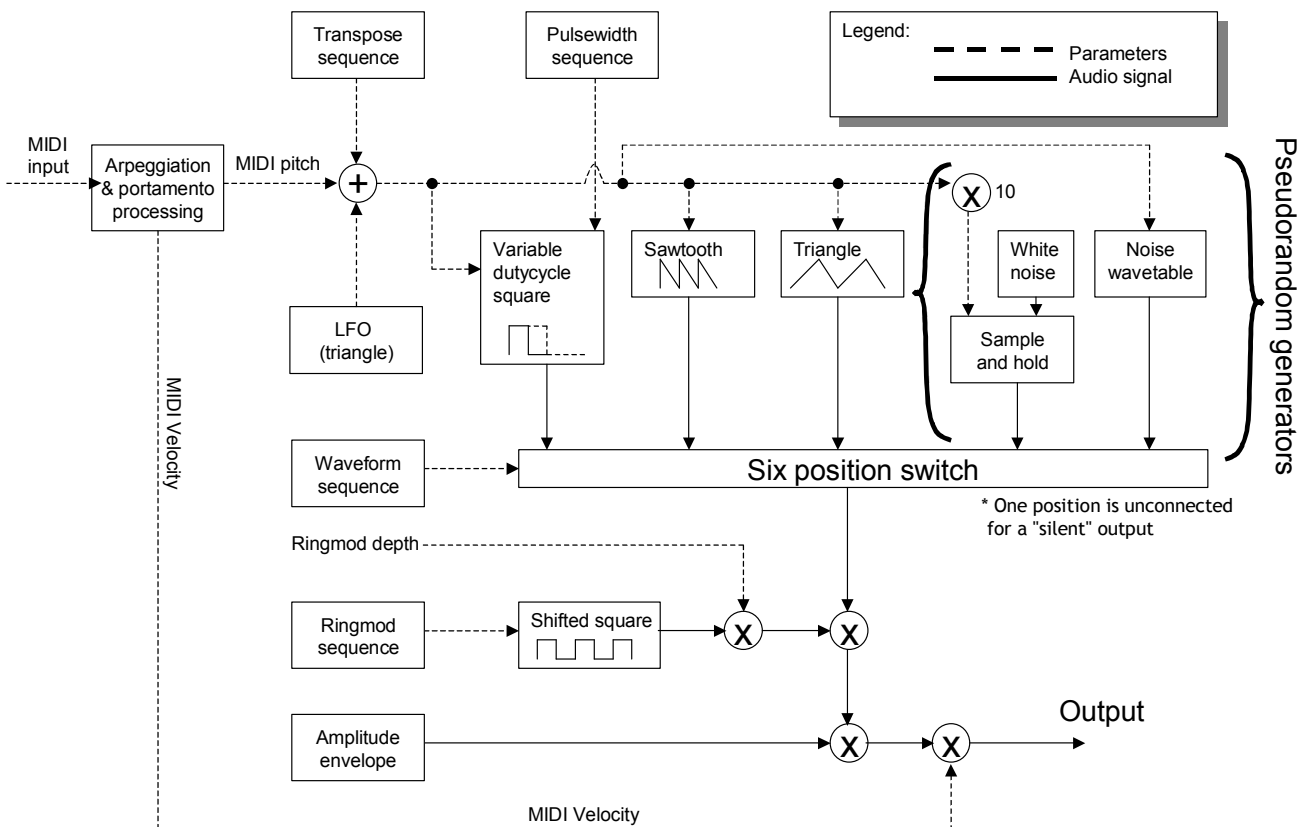


Figure 5.4 - Synthesis structure in overview

A bullet point explanation of Figure 5.4 is below.

- A number of oscillators are all set to oscillate at a specified frequency converted from a MIDI note pitch.

- This frequency is then offset by a transposition sequence
- This is also offset by an LFO with a triangle waveform
- One oscillator is a variable duty cycle squarewave, the pulse width is modulated by a pulse width sequence
- There are two pseudorandom value generators
 - The number of pseudorandom values per second in one “oscillator” is controlled by the clocking speed of a sample and hold unit.
 - The speed of the other pseudorandom value generator is controlled by the speed of playback of a wavetable
- The active waveform is selected by a 5-position switch, controlled by the waveform sequence
- This output is then ring modulated (amplitude modulation) by a squarewave oscillator (fixed 50% duty cycle), the frequency of this oscillator is controlled by a ringmod sequence.
- This output is then scaled by an amplitude envelope, and finally by the MIDI note velocity.

This is a very broad overview of the functionality; a number of the modules described above are made up of more complicated structures and certain things have been left out of this diagram to make it easier to get a picture of the basics. The conversion of MIDI note pitches to frequency values in Hz, the arpeggiation of chords, legato portamento, the structure of the sequenced parameters, and feedback of information to the user interface are expanded upon later.

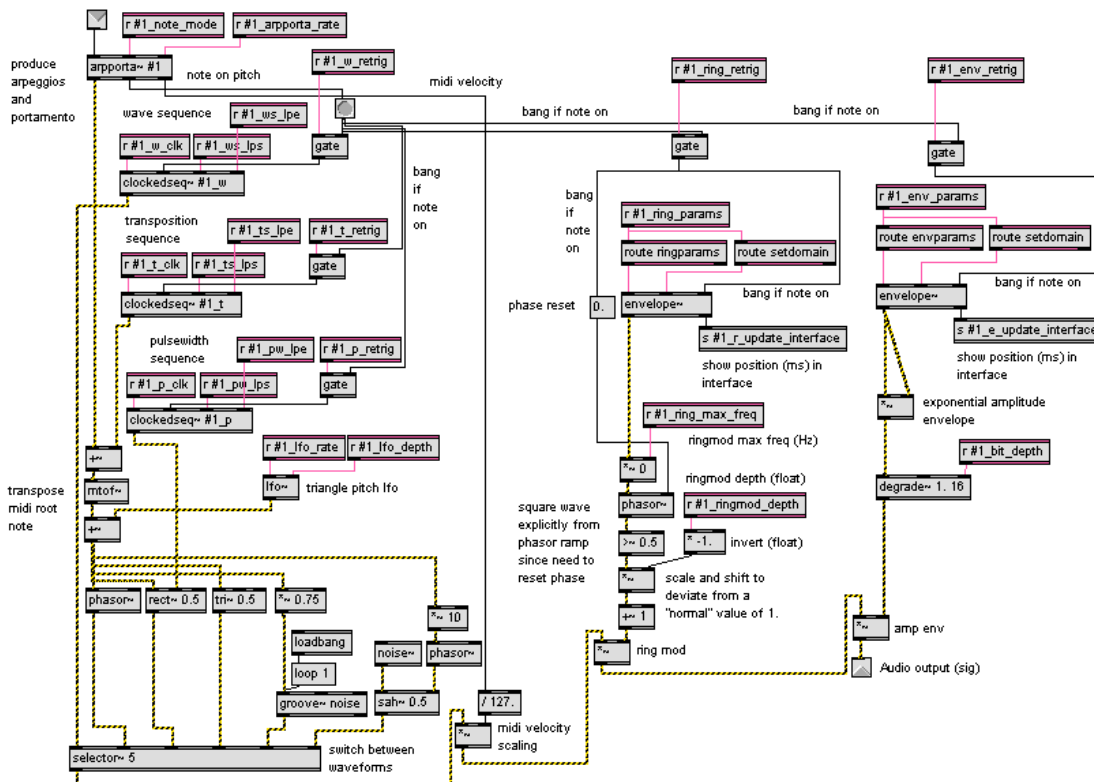


Figure 5.5 - Final MaxMSP synthesis structure

Figure 5.5 shows the final synthesis structure as it appears in MaxMSP. Again, this is quite complicated, so a comparison with Figure 5.4 on page 57 will be useful.

Some blocks and cables are labeled pink. The pink colouring was added by the author to indicate that the data flowing through these chains originates from the user interface. The pink chains have parallels with the hardware chip registers that were used to store synthesis parameters on the historic chips where information, defined at the user interface, eventually ended up being poked into registers, on the sound chips, to control the synthesis. Note that the top of the chain of all these pink blocks is an [r] object. Remember that this object is an abbreviation of [receive] and is paired with an [s] (an abbreviated [send]) object in the user interface. Note also that these [r] objects have an argument dereferencing symbol “#1” to obtain the argument in the calling abstraction. In most cases, this is the channel number of which the abstraction “belongs” to. Remember that this is used to prevent the multiple instances of abstractions in different channels from referencing the same data and influencing each other.

Comparing Figure 5.4 (page 57) with Figure 5.5, the ring modulation sequence and amplitude envelope are on the rightmost side of Figure 5.5. A long control-rate cable labelled “MIDI velocity” runs almost centrally, terminating with an a structure designed to scale the 7bit MIDI velocity values (integers 0-127) into a floating point range (0.0-1.0) that can be used to multiply the output from the oscillators providing amplitude velocity scaling.

To the lower left of this velocity scaling structure are the oscillators. The upper left holds the transpose, waveform, and pulse width sequences. The sequence abstraction is labelled [clockedseq~]. These abstractions are easy to identify in Figure 5.5 since they are populated by data entered at the user interface and so are connected to many pink objects. The “clock” part of this name is derived from the fact that the historic chips often updated the synthesis parameters by setting new values in the on chip registers by using interrupts that occurred at regular intervals.

It bears repeating that an abstraction is a patch file that is referenced by another patch to enable large and complex structures to appear as a single object in a max patch, reducing graphical clutter, and maximise code reuse.

In the case of the [clockedseq~] abstraction, a sequence of values is read from a buffer at specific intervals with the read pointer able to loop between specified minimum and maximum buffer index positions. In order to use its output to independently control properties of the synthesiser’s oscillator frequency, waveform, and pulse width for example, the structure within the [clockedseq~] must be laid down in the MaxMSP environment three times. This sort of duplication can make the number of objects in a patch grow very quickly. The use of an abstraction in this situation enables the reuse of this structure in an uncluttered manner.

Other custom “abstraction” objects include [lfo~]: a simple triangle lfo, [envelope~]: which was mentioned earlier used for the amplitude envelope and to control the ringmod frequency. Another custom “abstraction” is [arpporta~]: used to provide the modern features such as dynamic arpeggiation and portamento described in Section 2.3.2 and 3.3 (pages 35 and 46).

5.3.1 The [clockedseq~] abstraction

As mentioned before, the project started as a PureData prototype. When converting the prototype to a MaxMSP implementation, this was the first structure to be implemented and the last structure to be completed. It caused the most problems in development.

This is because it lies apart from pure synthesis technique. It represents and interacts with (in the context of the OSI model applied to chiptune synthesis in Table 1.3 on page 23) more layers than any other component of the system. Comparing it to historic systems, it “stands in” for the microcomputer’s interrupt and timing systems, for the system memory of the microcomputer that is used to store the sequence of values that will be inserted into the registers one by one, and for the interrupt service routines defined in the tracker software that will actually insert the values into the chip registers. In many ways it is the keystone of the entire system, since it is responsible for the rapidly changing character of the sound.

It mediates between the controls in the user interface, and the oscillators in the synthesis structure. Many revisions of the structure were made due to minor changes in this clocked sequence structure. A change to the way that the values were generated in the user interface necessitated a complete re-working of the way the values are converted before they are stored. This has a chain reaction; a change to the way that the values are stored necessitates a re-working of the way that the values are accessed and routed to the oscillator parameters.

This component therefore works hand in hand with other components in order to control synthesis parameters. It is not responsible for the specification of the values, that job belongs to the user interface. It is not responsible for the storage of values, that job belongs to another component. It is only responsible for the replaying of values, but even though the primary function of this component is to simply replay a stored sequence, the storage method of the sequence values must be known before this component can be designed.

The method of storage is discussed more fully in section 5.6 beginning on page 105.

Specification

The specification for this component is derived from the requirement outlined in section 4.3: “a state machine” like behaviour to “enable the user to specify sequences (or patterns) of synthesis parameter values that will be stepped through at a certain speed when a note trigger is received”.

This component must generate a signal-rate output, cycling through a sequence of values defined at the user interface. It must also produce a control-rate output indicating the current position in the sequence for feedback to the user interface.

The rate at which the numbers change (or the period for which each number is held) should be adjustable.

If each number in the sequence is assigned an “index” counting from zero to distinguish it from the number “value”; for example the sequence “5,1,9,0,3,8,4” would have indexes “0,1,2,3,4,5,6” so the number at index “3” is “0”. The component should provide a way of specifying the “index” at which the sequence repeats. For example, a sequence “5,1,2,1,8,3,6,9,5,4” may be replayed as “0,1,3,1,8,3,6,9,8,3,6,9,8...etc” by setting the

sequence to loop between index “4” and “7”. Numbers with indexes greater than seven are never outputted in this scenario.

It should be optional that a MIDI note trigger reset the sequence to the beginning (index “0”).

Options, Problems, and Decisions

This section is quite long; due to this object forming a core part of the sequenced parameter switching that is central to the sound character of chiptune timbres. The reader is encouraged to come back to this section in order to compare the final solution to the ideas discussed here.

As mentioned before, this component the storage method of the sequence values must be known before this component can be designed. See the “convertdata” abstraction for more information. In summary, the final solution was to store sequence values in a wavetable, or [buffer~] object.

This decision to store sequence data in a wavetable was influenced by the possibility of replaying the values in the wavetable at high speeds. Storing values in an object that only has control-rate outputs limits the interval between successive outputs to ~1 millisecond. The ability to control synthesis parameters with step sequences being played back at audio-rates opens a wide range of sonic possibilities enabling complex amplitude, frequency, and pulse width modulation techniques with fascinating interplay between fundamental frequencies of oscillators and the resulting sidebands in the frequency domain.

A related, very interesting, and potentially very powerful option considered was to take the idea of audio-rate sequenced parameters to its logical extreme by not fixing the destination of a given sequence of values.

Much to the author’s regret, this idea was not developed further or implemented in the final solution. The decision to abandon this approach represents an abrupt turning point for the entire project. The designs of many components were constructed in preparation for connection to a central matrix to control the routing of signal-rate modulation sequences. As explained, time constraints prevented this from being fully realised, and the internal structure of several components were altered to compensate. Echoes of the design philosophy behind this idea still remain in some component designs.

The conversion of [mslider] control-rate information to signal-rate [buffer] objects is strictly unnecessary in terms of the current specification, since the [clockedseq~] abstraction does not implement audio rate playback. If more time was available, the author would implement a way to use MIDI note pitch (possibly scaled by multiplying factor) to the playback speed of a step sequence driven by the custom [cindex~] external. This external is discussed in more detail in section 5.8, beginning on page 112.

This discussion of the unimplemented matrix control structure, hopefully explains why some components have unusual designs.

Compare Figure 5.4 where particular sequences refer to particular parameters, and the situation below in Figure 5.6.

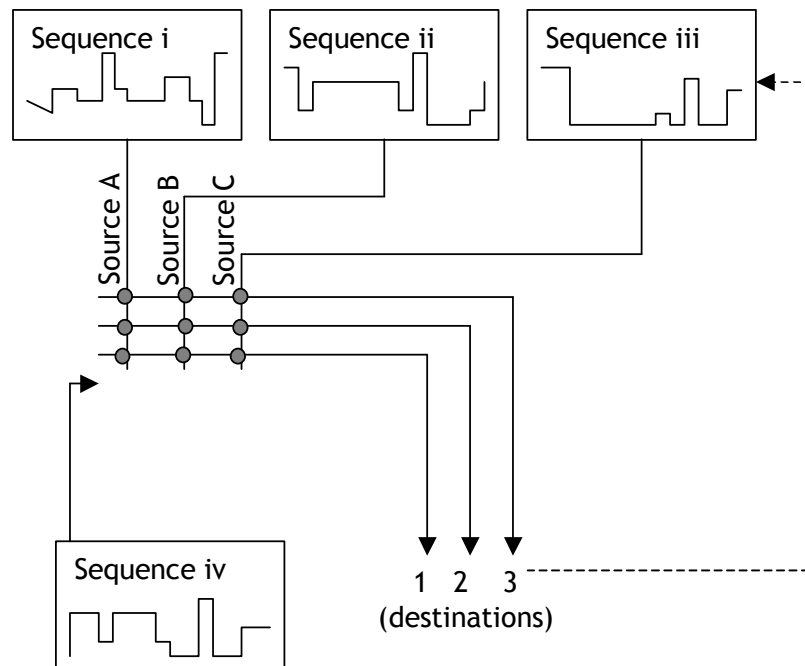


Figure 5.6 - Matrix control of parameter destinations with feedback path

Here, the source parameter sequences are not specifically assigned to modulate a fixed parameter but can be dynamically routed to destinations by means of a matrix that can be itself controlled by another sequence. The matrix output destination “1” may represent the active waveform, whilst “2” represents transposition, and a special case of destination 3 represented by a dashed line represents the data stored in “sequence iii”.

The result of implementing such a system would be a programmable synthesiser where every signal generator output could be used to modulate the parameters of any other signal generator. An extremely powerful consequence of this scenario is that by configuring the matrix to output a combination of sources including source C and routing the output to destination 3 (dashed line in Figure 5.6), the values in “sequence iii” could recursively influence themselves creating chaotic and unpredictable modulations.

If this idea were fully realised, the result would be a programmable modular synthesiser with a staggering number of configuration possibilities. This is perfectly suited to the style of audio programming environments such as MaxMSP and PureData.

The reader is encouraged to contrast the final solution to the ideas presented here and those discussed in section 5.7 on page 110, which discusses how an external object was developed to replace the graphically programmed version.

Leaving this idea aside, and focusing again on the first attempt at implementing the clocked sequence object, the approach was to mirror the method discovered in the research (section

3.2) whereby the rate at which an interrupt service routine transferred new values into the registers of the sound chips was controlled by one “master clock”, reconfiguring multiple sound chip registers simultaneously.

One of the options to discuss, therefore, is the possibility of breaking free of a “master clock”: providing individual clocking for each parameter that is to be controlled by a sequence. Another question might be: is it better to divide a master clock to produce unique clocking speeds per parameter or to implement a unique clock generator to provide unique clock speeds for each parameter sequence?

The benefits of the clock divider approach are that it relates very strongly to the historic systems, and so would likely provide a high level of compatibility with the characteristic sounds produced. It would be possible, for instance, to set up a transposition sequence of length 3, and a waveform sequence of length 2. A master clock would ensure that the clocks output in synchronism, and by setting their clock dividers to 3 and 4, each sequence loop will last the same length of time.

However with two independent clocks the user would be required to calculate the ratio of the clock speeds. The clock divider approach has its problems since (if integer divisions are used) it would not be possible to derive a clock speed between the master clock speed and half this value.

Moving on, one of the requirements for this component is that it produces a sequence of values that may loop between arbitrary points. In the example used previously: between index values “4” and “7”.

A block diagram of the required functionality (with oscilloscope traces at appropriate points) is below:

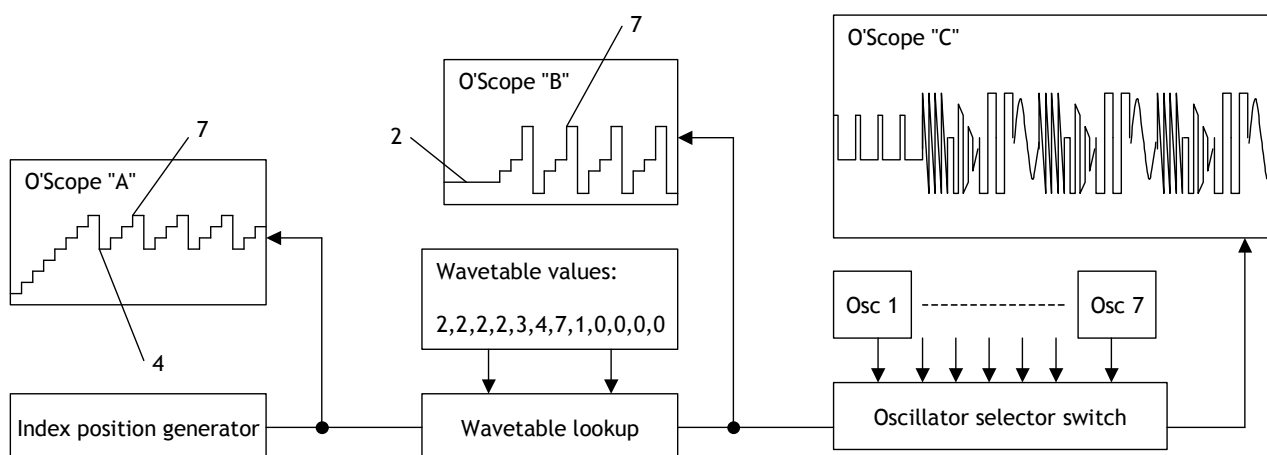


Figure 5.7 - Block diagram of intended functionality (sequenced waveform switching)

The oscilloscope trace “C” (top right) in Figure 5.7 shows the indented output waveform sequence. Note that the index position generator advances after a specific number of milliseconds so each waveform is held active for the length of time.

The first implementation of the sequence generator was to generate a ramp signal using a [phasor~] object, and to use the waveshaping [triangle~] object to scale the ramp amplitude between the minimum and maximum index values and to stretch the ramp period over the required number of index transitions. This method was awkward, using many graphically programmed objects to scale and shift values that were used to configure the [triangle~] and [phasor~] objects depending on the number of index positions. The benefits of this method were that it enabled playback of step sequences at audio-rates, opening the previously discussed range of powerful modulation techniques.

The second implementation was to use the [groove~] object, a wavetable replay object with the very desirable quality of an internal index position generator with the ability to skip back to a certain sample position. A typical application of the [groove~] object might be (in a sample based synthesiser) to replay a short sample from a recorded wind instrument such as a clarinet or flute. The [groove~] object would be configured to play back the sample from the beginning of the sample, but looping the last few cycles of the sample indefinitely, simulating the attack and sustained tone of the wind instrument with a very efficient use of memory.

Both the above implementations suffered from a problem, whereby the rate at which the control sequence is played back varied depending on how many control values were to be looped. Figure 5.8 is a screenshot from Cooledit (Syntrillium, 2002) illustrating how the gradient of the output ramp changes undesirably.

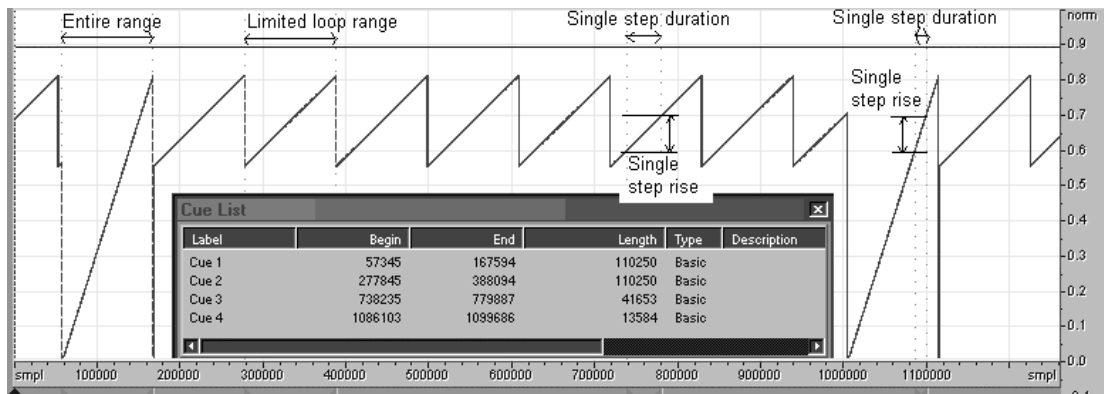


Figure 5.8 - Clock problems with looping wavetables

The gradient for the “entire range” ramp is steeper than that of the “limited loop range” ramp. The “cue list” window below is included to show that both ramps comprise of 110250 values. The pair of “single step duration” durations are different, the cue list measures them at 41653 values for a ramp in a limited loop range and 13584 values in the entire range. The result is that the sequence advances quickly if the entire range is to be output, and slowly for a small range.

The chosen solution was to use a control-rate [metro] object that advances a control-rate counter in discrete steps, and convert the output to signal-rate for use with the [index~] object. The rate at which the ramp advances is then divorced from the number of steps the ramp must advance. Figure 5.9 illustrates the resultant waveforms and proves that the gradients do not change undesirably as before.

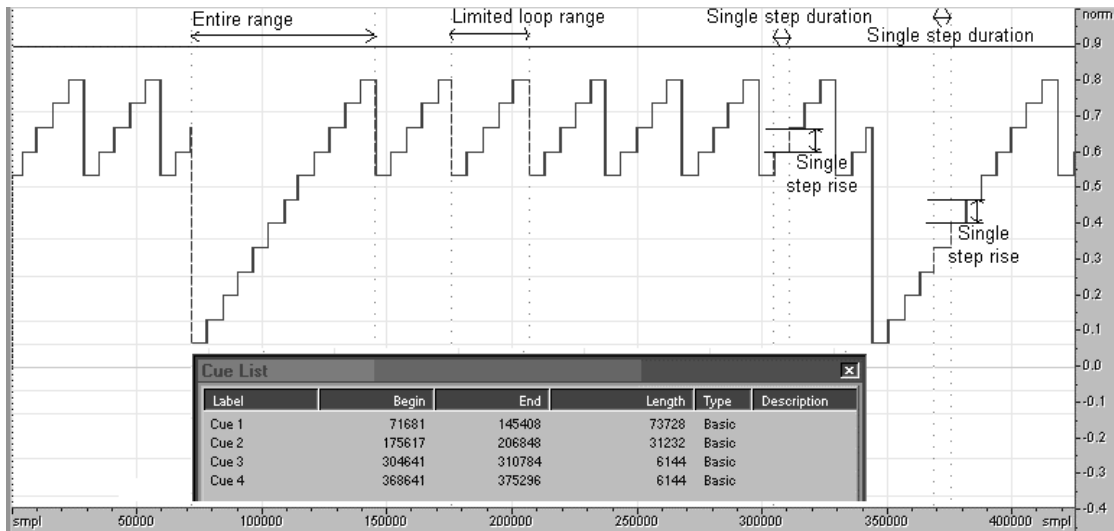


Figure 5.9 - Clock problems resolved using a [metro] and [counter]

Here, the corresponding “single step duration” for a “single step rise” is identical for both the “entire range” and the “limited loop range”.

The disadvantage of using this control-rate solution is that obviously, the playback of the step sequences is restricted to control-rates. With this method, it is no longer possible to use the step-sequences to provide complex audio-rate amplitude, frequency, and pulse width modulation effects.

Final solution

Figure 5.10 shows the final [clockedseq~] abstraction as it appears in MaxMSP.

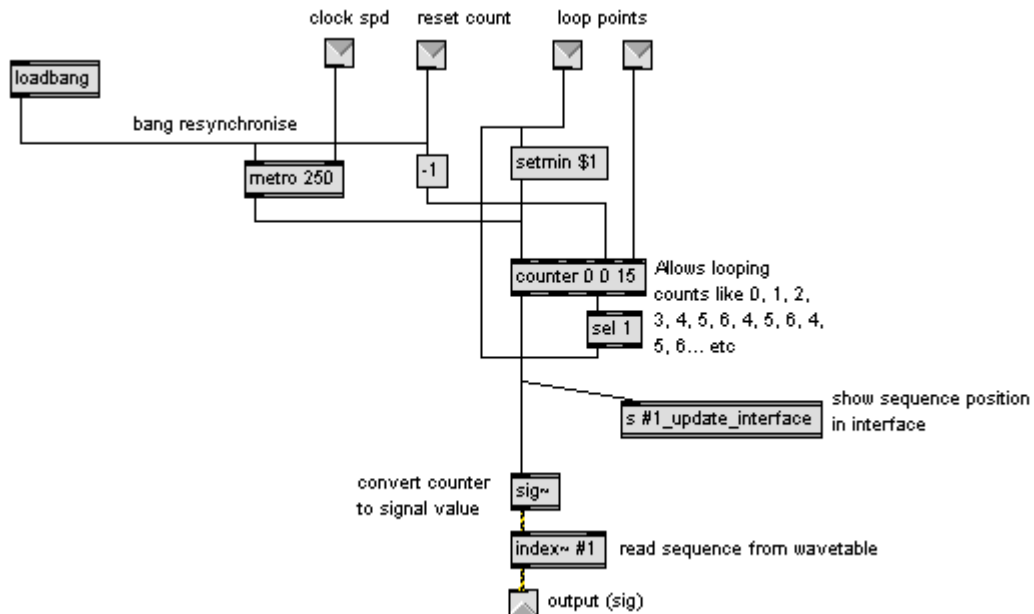


Figure 5.10 - Final clocked sequence structure

The central object (both conceptually and diagrammatically) in this structure is the [counter] object. This is a control-rate object that produces a series of increasing integer values. For

more information about [counter] see the MaxMSP reference manual. These integer values are used as index values to read from a wavetable holding a sequence of synthesis parameter values.

The [metro] object in the upper left increases the count at a defined interval (in milliseconds). The originator of this interval value is ultimately a number box in the user interface, Figure 5.5 shows that this value is passed to the metro object by a pink [r] object in the synthesis structure.

The structure allows for the counter to return to 0, this is used for retriggering the sequence on every new MIDI note. Also the minimum and maximum value the counter should produce can be set (these values also arrive by pink [r] objects, their values originating at the interface). Referring back to Figure 5.5, [gate] objects are used to block or pass the instructions to reset to zero every time a MIDI note on message is received. In this way, some sequences can be set to loop forever, whilst others may “retrigger” for each MIDI note received.

A feedback mechanism allows the counter to be configured to count from 0, advancing every “n” milliseconds, whereupon reaching a specified value “max” it returns to a specified value “min”. This structure allows a small section of the entire sequence to be looped as explained above, say from index “4” to index “7”.

The control-rate output of the counter is fed to an [s] object that the user interface uses to tell the user of the current index position. The output of the counter is also converted to signal-rate and used to feed the [index~] object which reads from the specified wavetable.

The wavetable is specified as an argument passed to the [index~] object. Remember that the “#1” token represents the first argument passed to the abstraction. In this way, unique index objects read from unique wavetables for each synthesis channel.

Dependencies on other components

The component relies on data being converted from the user interface into a wavetable by the [convertdata] abstraction.

It also relies on the user interface values being correctly sent to the pink [r] objects in the synthesis structure parent abstraction. These values are converted from the user interface by the [transportdata] abstraction.

Testing notes

Since the abstraction is built using objects documented in the MaxMSP reference manual, the boundary conditions for the input values are known. Various values for clock speed, and loop positions were tested both inside and outside the range of values specified in the MaxMSP reference manual in order to identify any problems that may need to be addressed by setting limits in the user interface input values for example.

When a maximum loop position less than the minimum loop position is set, the counter does not advance past the lower of the two values. The user interface does not warn the user of this situation, but does limit the range of possible minimum and maximum positions between zero and 16 to prevent specifying index positions outside of the wavetable length.

This enforced limit of 16 steps seems an adequate length for the step sequences, for more demanding sequences it could be expanded very easily with modification to the [convertdata] abstraction and user interface components. In fact, the user interface object [mslider] (discussed later) supports dynamic resizing of parameter data, opening the possibility of allowing the user to choose the size of the sequence and simply passing appropriate values to the [buffer~] objects that allocate wavetable memory, and to the minimum and maximum [counter] objects in the [convertdata] and [clockedseq~] abstractions.

5.3.2 Arpeggiation and Portamento [arporta~] abstraction

This section discusses the automatic arpeggiation and portamento processing of polyphonic MIDI input. The inspiration for these ideas is outlined in section 3.3.

Options, Problems, and Decisions

In early implementations of the system, arpeggiation could only be achieved by setting an appropriate transposition sequence. Monophonic input would be arpeggiated by the cycling of values in the transposition sequence as opposed to a dedicated module processing the incoming MIDI data.

This method was limited in terms of the harmonising of the resultant arpeggiations. For example: a transposition sequence with three values: “0, 4, 7” equates to a chord containing a root, a major 3rd (4 semitones interval), and a 5th (7 semitones interval).

If the root note is C, this will produce an arpeggio of three notes: “C, E, G” which is a major triad in the key of C. If this root note (C) is itself transposed by 2 semitones to D, the resulting transposition will produce notes “D, F#, A”. These notes correspond to the root (D), a major 3rd (4 semitones away from “D” is “F#”), and a 5th (7 semitones away from “D” is “A”).

Both root notes (C and D) are in the key of C, yet the resultant arpeggiations (due to the method by which they are derived) include notes that are outside this key of C. Figure 5.11 explains this graphically using music notation and a representation of a MIDI keyboard.

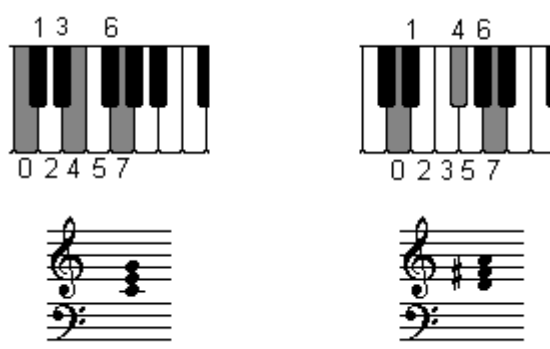


Figure 5.11 - Transposition problems with conflicting keys

The inclusion of “F#” could be a potential problem, since it is not in the key of C. The process is correct: it produces a major triad in the key of the root note, but it does not restrict the voicing to notes in a specific key. In terms of this automatic arpeggiation being used in the context of a certain musical key, it is unusable.

A solution to this problem might be to have a user defined “desired musical key” input fed to a structure that mathematically adjusted the transposition sequence, restricting or shifting the transposed notes to lie within the desired key. Another alternative would be to use manually identify clashes and to use “zones” (as described in section 2.3.2) to activate alternative transposition sequences.

The final solution does not try to use the “transposition sequence” method to generate arpeggiations. This was abandoned in favour of a dedicated module for creating portamento (note slides) in addition to arpeggiations.

Specification

This component should detect polyphonic MIDI input in terms of sequential note pitches and velocities, and output corresponding monophonic MIDI note pitches and velocity pairs in one of two modes:

1. In arpeggiation mode, it should output a stream of note on/off message pairs at a specified and adjustable interval in milliseconds. The note pitches should be sorted into ascending pitch, and only one note pitch should be “on” at any one time.

When a note off message is received, all notes should be silenced.

Note velocities are paired with their input pitch and output accordingly. An input with three pitches, each with a unique velocity will be arpeggiated with the appropriate velocity for the input pitch.

2. In portamento mode, in the case of a monophonic input, the messages should pass through unaltered. In the case of a polyphonic input, the “oldest” note pitch should be used as the starting pitch for a linearly interpolated “glide” or portamento to the “newest” note pitch. The interpolation should take place over a specified and adjustable interval in milliseconds.

When a note off message is received, all notes should be silenced.

Note velocity is constrained to the first velocity input.

This style of portamento is often described as a “legato portamento”.

3. In bypass mode, the input is mirrored to the output, polyphonic note on messages are converted to monophonic output by way of discarding the “oldest” note pitch and velocity, replacing the old values with the most recently received.

Inputs

Referring back to Figure 5.5 on page 58 which shows the MaxMSP synthesis structure, the abstraction is placed in the top left corner where two pink [r] objects configure the object based on values originating at the user interface.

The inputs to the [arpporta~] abstraction are:

- MIDI data, packed integer message (note pitch first, velocity second)

- Note mode, integer control-rate:
 - 0 = direct bypass,
 - 1 = arpeggiator,
 - 2 = portamento
- Arpeggiation/Portamento, integer control-rate:
 - Arpeggiation: milliseconds interval between notes
 - Portamento: milliseconds duration of slide between notes

Outputs

The abstraction produces three outputs:

- A signal rate value between 0 and 127 which represents the MIDI note pitch
- A control-rate integer MIDI note pitch, for use in detecting unique note pitches in other components
- A control-rate integer MIDI note velocity, for use in velocity scaling amplitude

Final Solution

Figure 5.2 on page 55 depicts the [arpporta~] abstraction as a single module, with all functionality contained within this one layer. There are in fact two further abstractions contained within the [arpporta~] abstraction: the first is [midiarpeggiator~], the second is [midiportamento~]. Figure 5.12 shows a block diagram of the structure including some internal components of the sub-abstractions, whilst Figure 5.13 shows the final MaxMSP structure.

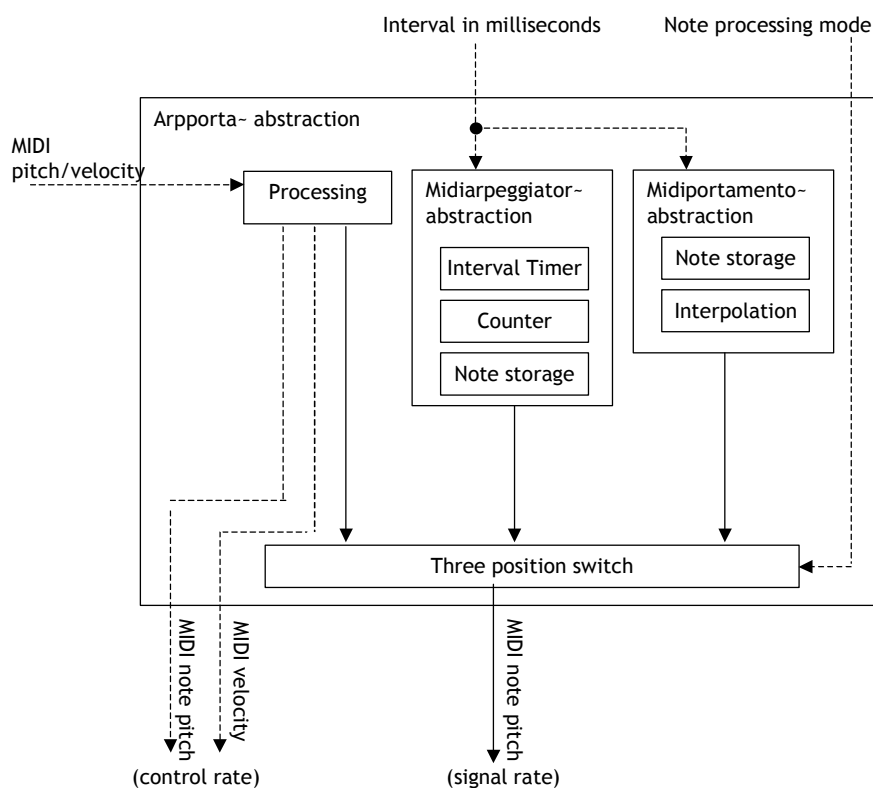


Figure 5.12 - Block diagram of arpporta~ abstraction functionality

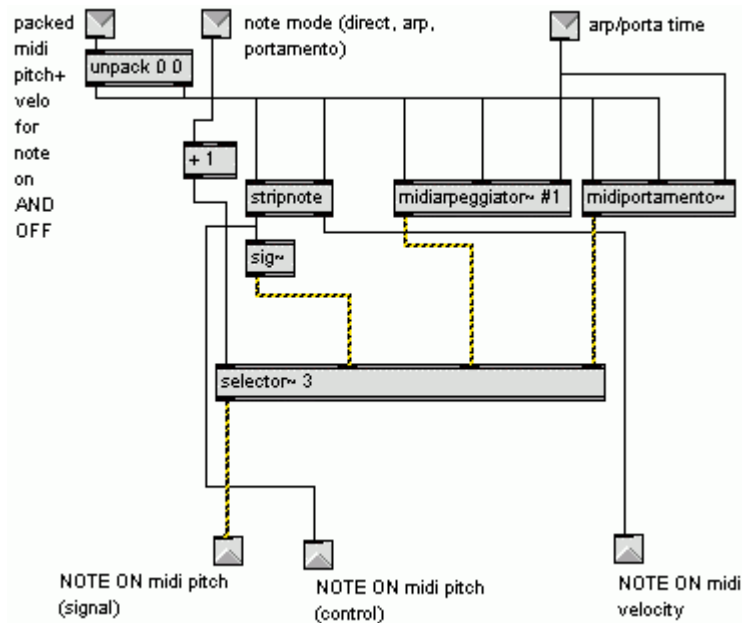


Figure 5.13 - Final arpeggiation and portamento MaxMSP structure

Dependencies on other components

This component relies on the inclusion of the [midiarpeggiator~] and [midiportamento~] components for its principle functionality.

Testing notes

Consequences of using velocity sensitive keyboard are potentially musically expressive strange jumps in amplitude?

testing mainly of sub-components described later.

5.3.3 The [midiarpeggiator~] abstraction

This abstraction is a sub-component of the [arpporta~] abstraction.

Specification

This component must store information about the currently held down notes, turning polyphonic input into arpeggiated monophonic triggers.

If just one note is held down, it must output a note on message only once, this will avoid unnecessary stuttering of monophonic input.

If more than one note is held down, the currently held note pitches should be cycled, producing note on (and appropriate note off) messages at a certain specifiable and alterable interval.

Options, Problems, and Decisions

A desirable feature is that when a “note off” message is received for a MIDI note, this note should no longer occur at the output until a “note on” message for this note is received

again. In this way, it will be possible to start with an arpeggiation sequence with a large number of unique pitches, gradually removing notes from the arpeggiated chord.

The interval between the notes of an arpeggiated chord should remain fixed, regardless of how many notes are in the chord. This would involve dynamically altering the note duration, depending on how many simultaneously held notes are detected at the input.

An experimental version of this module used wavetable storage using the [buffer~] object. This method was initially chosen in order to be compatible with the matrix routing of signal-rate control values discussed in Figure 5.6. By using a wavetable instead of control-rate objects, the arpeggiation sequence could potentially be replayed at audio-rates producing (no doubt) highly unusual frequency modulation effects as a result of the stepped modulation pattern.

This wavetable [buffer~] storage solution was abandoned since removing a specific note pitch from a list of values stored in a wavetable when a “note off” message is received is extremely difficult. The wavetable must be searched by index for the required pitch value, the sample must be “deleted”, and all subsequent samples must be shuffled back to alter the length of the sequence.

Due to this problem, the wavetable approach was abandoned in favour of using the control-rate [coll] object for storing the list of MIDI note pitches. The [coll] object implements a very efficient and fast method for removal of specific entries and resorting of the list, but due to time restrictions this feature is unused in the project. When a “note off” is received, the entire arpeggiation list is cleared.

Final solution

Shows the final MaxMSP structure.

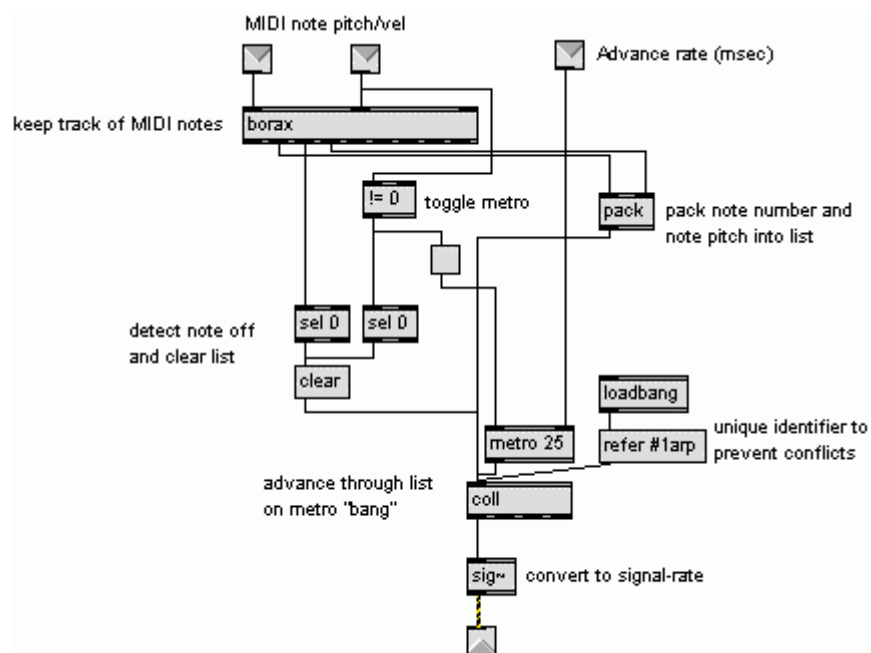


Figure 5.14 - Final MIDI arpeggiator MaxMSP structure

The core functionality for this abstraction is provided by the [borax] object. This object keeps track of MIDI note pitches, counting the number of currently “held down” notes assigning a “voice number” based on arrival time. This information is stored in a [coll] object and accessed with a [metro] object, as previously explained.

When a note off message is received, or the number of “held down” notes equals zero; the list of notes held in the [coll] is cleared, and arpeggiation is stopped.

For single note pitches, the [coll] is populated with only one value. Since only the MIDI pitch information is arpeggiated for each note (i.e. the velocity information is not arpeggiated) the arpeggiator does not cause a re-triggering of the amplitude envelope elsewhere in the synthesis structure. The result is that single notes do not cause stuttering when arpeggiated.

Inputs

Referring back to Figure 5.13 on page 70 which shows the [arpporta~] abstraction, the three inputs to [midiarpeggiator~] are:

- MIDI note pitch
- MIDI velocity
- Arpeggiation rate (integer control-rate in milliseconds)

Outputs

- MIDI note pitch (signal-rate)

Dependencies on other components

The principal functionality of this abstraction depends on the [borax] object. Changes to the way that this standard MaxMSP object functions may require modification to this abstraction.

This object references depends on the MIDI pitch, velocity to be passed from the parent [arpporta~] abstraction. The arpeggiation rate is defined at the user interface and must be correctly passed through pink [s] and [r] objects in the [transportdata] abstraction and synthesis structure.

Testing notes

In the testing stages, the object sometimes behaved unexpectedly when the user interface switched note modes between “direct” and “arpeggiation”. The [coll] object was being populated by the notes received in “direct” mode and when switching to arpeggiation, the [coll] was not empty. Also MIDI input featuring staccato chords sometimes resulted in stuck notes, where the [borax] object still registered a “held down” note even though there was none.

To prevent this behaviour, the [coll] is cleared every time a “note off” message is received and when the note mode is switched to ensure that the arpeggiation list is empty.

The arpeggiation rate fed to the [metro] object is constrained in the user interface to a range no shorter than 2 milliseconds. No upper limit is enforced.

5.3.4 The [midiportamento~] abstraction

This is a sub-component of the [arpporta~] abstraction.

Specification

This component should provide a legato portamento signal-rate MIDI note pitch. That is to say it should jump instantly to new positions if a “note off” is received before a new “note on”. Otherwise (if notes are “slurred” or held down between changes in pitch) the MIDI note pitch should interpolate (glide) between the held note pitches at a specified rate in milliseconds.

In the case of a monophonic input, the messages should pass through unaltered, single notes do not trigger a note slide, but jump instantly to the note pitch. In the case of a polyphonic input, the “oldest” note pitch should be used as the starting pitch for a linearly interpolated “glide” or portamento to the “newest” note pitch. The interpolation should take place over a specified and adjustable interval in milliseconds.

When a note off message is received, all notes should be silenced.

Note velocity is constrained to the first velocity input.

Options, Problems, and Decisions

Very few problems were encountered in the construction of this abstraction.

Some decisions had to be made regarding the interpolation method between notes. For example, deciding between a linear or an exponential glide between notes, and deciding whether to adjust the rate at which the note pitch slides between pitches to ensure the same number of Hz per millisecond.

Very early experimental versions of this abstraction used control-rate signals for interpolation, converting to signal-rate right before the output. This produced “zipper” stepping of the interpolated pitch ramp due to the crude time resolution of control-rate signals.

The final solution uses signal-rate linear interpolation, and makes no adjustment to the time taken to slide, the time taken to glide is absolute regardless of the size of the interval across which the slide takes place.

Final solution

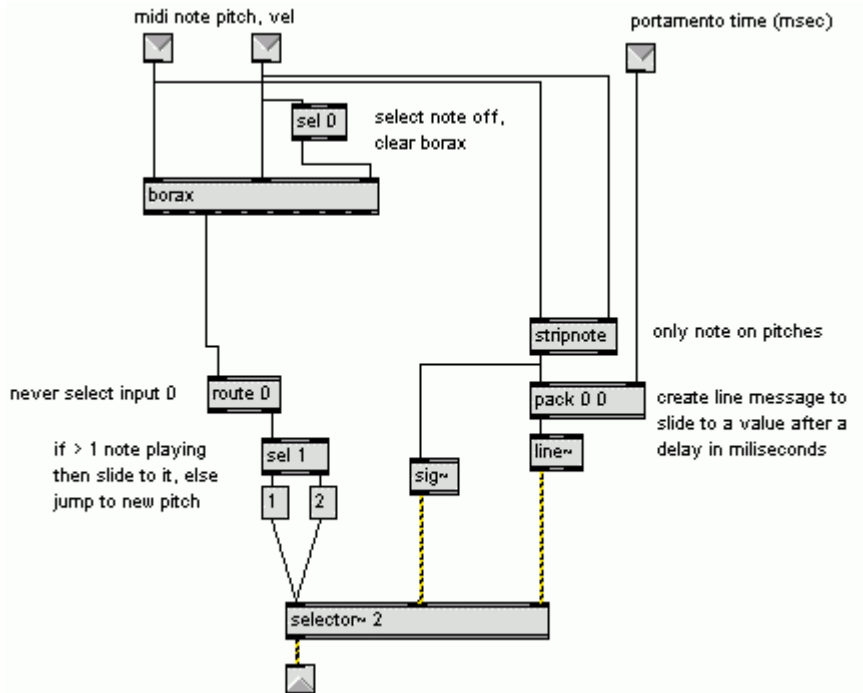


Figure 5.15 - Final MIDI portamento MaxMSP structure

As with the [midiarpeggiator~] abstraction, the key object here is the [borax] object, used to determine if more than one note is held down.

For single notes, the [stripnote] object is used to return the current “note on” pitch. Early experimental versions of the abstraction did not account for this and produced strange portamento behaviour whereby “note off” messages would cause slides to happen undesirably.

The [selector~] object switches between the signal-rate interpolated pitch and the direct pitch derived from the [stripnote] and [sig~] objects.

The [route] object in the lower left of Figure 5.15 ensures that the [selector~] object is never “turned off” when no MIDI input is detected and [borax] reports zero held down notes.

Inputs

Referring back to Figure 5.13 which shows the [arpporta~] abstraction, the three inputs to [midiportamento~] are:

- MIDI note pitch
- MIDI velocity
- Portamento rate (integer control-rate in milliseconds)

Outputs

- MIDI note pitch (signal-rate)

Dependencies on other components

The principal functionality of this abstraction depends on the [borax] and [line~] objects. Changes to the way that these standard MaxMSP objects behave may require modification to this abstraction.

This object references depends on the MIDI pitch, velocity to be passed from the parent [arpporta~] abstraction. The portamento rate is defined at the user interface and must be correctly passed through pink [s] and [r] objects in the [transportdata] abstraction and synthesis structure.

Testing notes

Like the [midiarpeggiator~] abstraction discussed in section 5.3.3 (beginning on page 70) the velocity of the MIDI input is not processed by the abstraction. Using a velocity sensitive MIDI input keyboard to play sounds featuring portamento result in musical phrases that can be extremely expressive.

By playing alternate quiet and loud notes in a legato style with a fast attack amplitude envelope, the output will jump between amplitudes and slide between pitches. This is a very unique behaviour that was unspecified and unexpected, but very welcome!

5.3.5 The [lfo~] abstraction

This is a tiny abstraction created to avoid clutter in the synthesis structure. It is extremely trivial and is mentioned here for completeness.

Options, problems, and decisions

One of the main options is the shape of the LFO waveform. As discussed in more detail later in section 5.3.6 on page 77, the original method of generating a triangle waveform was to mathematically shift and scale a saw tooth ramp. This method was later replaced by using the more efficient [tri~] object.

The possibility of allowing different waveshapes was considered, possibly by using a wavetable with an optionally interpolated (as opposed to discrete steps) step sequence. This idea was initially chosen to stay in keeping with routing of signal-rate control signals through a matrix as discussed in Figure 5.6. This would be an extremely powerful modulation tool if interpolation and an audio rate output routable to many parameter destinations were available. An alternative to this powerful modulation matrix approach would be to allow the user to choose from a limited selection of pre-defined waveshapes and pre-defined modulation destinations.

Of particular relevance to pitch modulation: a delay could be added to the LFO signal, enabling a note to begin at a steady fixed pitch, gradually developing a vibrato.

Another enhancement might be to provide a velocity sensitive rate and depth control, allowing the user to cause wild variations if notes are struck with a certain velocity. A unique addition would be to specify the sign (negative or positive) of the sensitivity so that a wild variation could be specified for notes with low velocities, and zero variation for high velocity notes.

The final solution specification does not include any of these more advanced features due to time constraints, however; the modular approach to the construction of the project makes the addition of new features at a later stage relatively easy.

Specification

This component should produce a low frequency oscillation as a signal-rate control signal.

The frequency and amplitude range of the oscillation should be specifiable and alterable.

An LFO intended for tremolo effects (amplitude offset) should only output positive values. The output signal for this LFO however, is intended for use as a frequency offset for vibrato effects. The LFO output can therefore swing negative without any ill effects.

Final solution

As mentioned earlier, the final solution is trivial and is mentioned here for completeness.

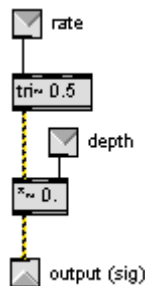


Figure 5.16 - Final lfo MaxMSP structure

Inputs

- LFO frequency: floating point control-rate (Hz)
- LFO amplitude: floating point control-rate

Outputs

- Oscillation: signal-rate

Dependencies on other components

The abstraction depends on the frequency and amplitude values being passed correctly from the user interface through the [transportdata] abstraction through [r] and [s] objects to the synthesis structure.

Test notes

Specifying a negative value for either frequency or amplitude does not affect the perceived result. A zero frequency value pauses the output of the oscillator, a zero amplitude value obviously silences the output.

Since negative values do not affect the perceived output, the constraints placed on the inputs in the user interface are purely to prevent user confusion. Positive (including zero) values are permitted for both input parameters.

5.3.6 Oscillators

Options, Problems, and Decisions

Various different configurations of oscillators were considered. Particular experimentation was devoted to different ways ring modulation could be achieved. As explained in section 3.1, the Commodore SID chip featured a matrix system similar to that described in Figure 5.6 (page 62) but for oscillator outputs instead of control sequences. The result is that each SID channel could be configured to ring modulate any other (but not itself). This particular technique was not duplicated; instead a dedicated oscillator is included for each channel as will be explained later.

The author considered the possibility of extending the matrix system described in Figure 5.6 (page 62) where audio outputs of channels could be routed to parameter sequence buffers, creating an extremely flexible and potentially complex system of data flow: A parameter sequence could be routed to the oscillator waveshape and frequency parameter simultaneously, the output of this oscillator could be routed to this parameter sequence creating an extremely strange frequency modulated setup no doubt with extremely interesting sonic results! Again, this is something that was not implemented but that the author would like to pursue at a later stage.

Another area of development that was developed significantly is the pseudorandom noise generator. A popular method of generating digital pseudorandom noise is to use a shift register and a number of exclusive or (XOR) gates in a feedback arrangement. This arrangement will always cause the sequence of numbers outputted by the shift register to repeat. By altering the positions at which the XOR inputs taken from various positions along the shift register, the length of the repeating pattern can be altered. If the shift register is repeating its sequence at audio rates, the output will be perceived as a pitched buzzing or rasping tone. The Atari TIA chip, and the Nintendo 2A03 (and GameBoy variants) use this technique in combination with adjusting the clocking rate of the shift register to generate “pitched noise tones”, to adjust the pitch of the buzzing or rasping at musical intervals.

Inspiration was also taken from the POKEY chip, where the output of a noise generator is fed to a “sample and hold” unit triggered by an oscillator. Sweeping the oscillator frequency whilst listening to the output of the sample and hold unit creates a very distinctive pattern of spectral content that is particularly desirable for chiptune music. POKEY chips were used in a large number of arcade machines, making this style of white noise generation into many chiptune compositions for arcade games.

Another option would be to implement a “Galway noise” generator. This technique is attributed to Martin Galway in an interview (Tuwien.ac.at, 2007), and is implemented on the commodore SID chip by abusing leakage currents in the volume register. The bulk of the work is done by the microprocessor (rather than a hardware oscillator) with parameter values being sent to the SID chip at audio rates, creating a train of clicks and pops which are perceived as a tone.

This technique is recreated in software by the QuadraSID software synthesiser (Refx.net, 2007). Analysis of waveforms by the author show that the basic principle is a modulo 16 counter, incrementing the volume registers by a value of “n” after a pause. A parameter specifies the number of times to increment by that value (in modulo 16 arithmetic), another

parameter determines the pause (in clock cycles) after incrementing, and a further parameter determines the pause before moving on to next step in a sequence.

Oscillator techniques

During development, different synthesis techniques for generating square, triangle, and sample-and-hold white noise were tried out.

Only one oscillator waveform is to be active at any one time, so the initial implementation was to use a waveshaping approach where a master oscillator provides the basic periodic oscillation and the required waveshapes are derived by mathematical manipulation.

Early revisions of the oscillator structure generated a master oscillator ramp using a [phasor~] object and distorted the shape of the ramp into a triangle wave, and squarewave. A triangle wave was derived by shifting and inverting the ramp after it exceeded a certain value. A variable duty cycle squarewave was derived by taking the output of an [>~] object, the larger the comparison value, the shorter the duty cycle.

These methods are not “band-limited”, and produce pure mathematical waveshapes with harmonics that extend beyond the sample rate. These cause strange aliasing effects when the master [phasor~] is set to a high frequency. The final solution employs objects that generate efficiently band-limited square, and triangle waveshapes without using a master [phasor~] oscillator. This has the advantage of being more efficient in the long run than generating (for example) a triangle waveform by mathematically processing a ramp signal.

MaxMSP includes an object to generate band-limited sampled-and-held noise. This object was deliberately avoided since aliasing harmonics are part of the desirable character of this particular sound. Band-limited noise at low sample rates sounds dull and obviously filtered, losing some of the characteristic grittiness.

The “pitched noise” oscillations commonly implemented by a shift register with XOR feedback taps as explained previously, is simulated by using a [groove] object that provides looping wavetable playback with no interpolation between samples. The author found that using interpolation of any sort smoothed the characteristic grittiness, in an undesirable way, similar in manner to that of band-limiting the sampled noise.

Waveform switching

Several options were considered in terms of waveform switching. One option would be to write a specialised external object that would accept signal-rate control values that would determine the frequency of the oscillation, and the active waveform. Another option considered, was to have a single master oscillator wavetable lookup oscillator, and simply switch the wavetable that it accesses. Either method would increase computational efficiency.

From Figure 5.17 on page 79 and from Figure 5.5 on page 58, it can be seen that the final MaxMSP structure uses a [selector~] object to switch between the oscillator outputs, the switching is driven by the [clockedseq~] abstraction to set the waveform.

The [selector~] object switches between different input signals in a sample-accurate manner, this can produce clicks in the output if the inputs being switched have differing amplitude values between adjacent samples.

The [cleanselctor~] object by Eric Lyon (2006) is an object that “gives clean crossfades between signal sources”. The use of this object was considered, but due to time restrictions the inclusion of cross fading between active oscillators was not added. This functionality would be extremely easy to add in a future version of the synthesiser.

Specification

Section 4.3 (page 49) describes the required oscillator waveforms and the reasons for including each in detail. The waveform list is summarised below:

- Sawtooth oscillator
- Square (variable pulse width) oscillator
- Triangle oscillator
- Periodic (pitched noise tone)
- Non-periodic (sampled) white noise

In addition to these waveforms a “silent” output should also be possible, enabling stuttering or echo effects to be created by swapping between silence and another waveshape.

Final solution

The final oscillator structure is very simple and can be seen reproduced here for convenience. The reader is strongly encouraged to view the full synthesis structure of Figure 5.5 on page 58 to review the context.

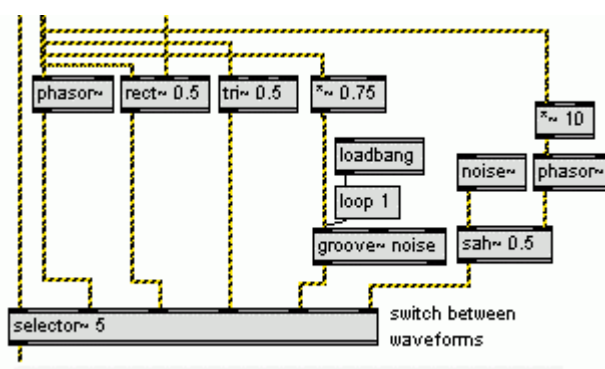


Figure 5.17 - Isolated oscillator MaxMSP structure

Here, multiple oscillator waveshapes are computed in parallel. This method is not very efficient in terms of computation, as discussed earlier.

Dependencies on other components

The oscillator structure depends on the most number of abstractions in the entire system. Parameters supplied by several [clockedseq~] abstractions, the [arpporta~] abstraction (including the sub-components [midiarpeggiator~] and [midiportamento~]) and the [lfo~] abstraction are all required for correct operation. These abstractions also depend on parameters supplied to these components by the [convertdata] and [transportdata] abstractions, and on appropriate values originating from the user interface via [s] and [r] objects.

The sampled noise structure depends on the [sah~] and [noise~] objects.

Testing notes

As described in the previous paragraph, the structure depends on a great deal of other components; the testing methods and results for all of these components influence the behaviour of the oscillator structure. Rather than repeating these test notes here, the reader is encouraged to review the testing notes for these components.

5.3.7 Amplitude envelope structure

This structure is part of the [synth~] abstraction. A sub component is the [envelope~] abstraction.

Options, Problems, and Decisions

The amplitude registers of the Atari YM2149 (a clone of the General Instruments AY-3-8910) featured logarithmic attenuation of oscillator outputs that complement the human hearing sensitivity. The 4-bit low resolution of the logarithmic DAC creates a characteristic effect for pad sounds with slow attack or release amplitude envelope phases.

The decision was made that this logarithmic behaviour would be recreated, but the user should be presented with the option of altering the amplitude resolution to higher or lower values between one and 16 bits.

Another option would be to implement some version of a Galway noise generator here. Section 5.3.6 discusses this in more detail. The decision not to implement a Galway noise generator here was enforced by the decision not to use a step sequencer and/or counter to control the amplitude. See also section 5.3.9 (page 85) for more detail.

Early versions of the amplitude envelope employed a [degrade~] object to quantise the amplitude resolution between one and 16 bits, in conjunction with a [pow~] object to create the logarithmic scaling of the signal to the base “e” (i.e. ~ 2.718). When testing the efficiency of the system it was discovered that the [pow~] object is less efficient than simply squaring the values. The range of input values from the [line~] object does not exceed 1.0 so the values may safely be squared without fear of clipping the output signal.

Specification

This structure must accept parameters fed from the user interface, primarily that of data from a breakpoint [function] object, and be able to trigger the envelope shape every time a “note on” trigger is received. This functionality is mainly achieved by the [envelope~] abstraction (discussed later).

The gain adjustment must have a logarithmic amplitude scale.

The amplitude of the envelope should be quantised to a specifiable and alterable degree.

The logarithmic, quantised, ramp signal should be used to adjust the gain of the main oscillator signal.

Envelope duration “setdomain” configures scaling outside in interface, and user feedback ramp.

Final solution

For convenience, the amplitude envelope structure is repeated below. The reader is strongly encouraged to view the entire synthesis structure in Figure 5.5 on page 58 to help place this structure in context.

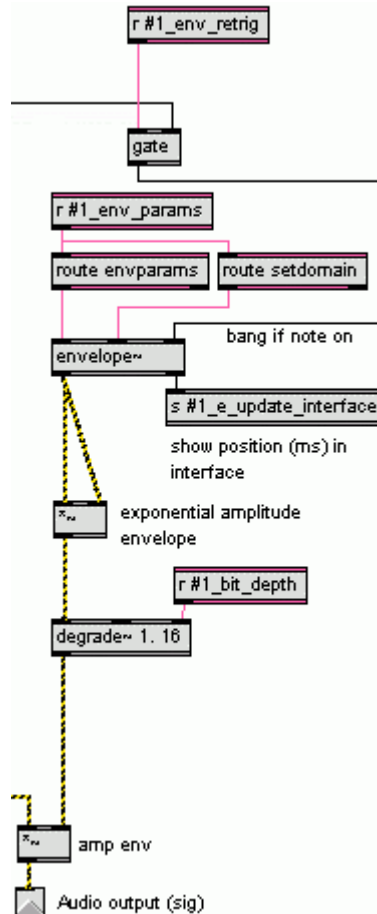


Figure 5.18 - Final amplitude envelope MaxMSP structure

As described before, the “#1” token is used again to access the 1st argument passed to the synthesis abstraction.

The pink [r] objects receive messages from the user interface. The [route] object separates out the data from the [function] object, and the data intended for the control-rate user feedback ramp to indicate the position of the envelope in milliseconds (see section 5.3.9 on the [envelope~] abstraction). This ramp is routed to the user interface by an [s] object.

The [gate] object at the top of the structure allows for optional retriggering of the envelope for “note on” triggers.

Inputs

- Envelope parameters from user interface as [r] object
 - “All points in line format” preceded by the string “envparams”
 - Envelope duration (control-rate integer) preceded by string “setdomain”
- “Note on” trigger: control-rate bang message
- Retrigger on “note on”: control-rate integer

- Amplitude quantisation bit depth: control-rate integer
- Input signal

Outputs

- Gain adjusted oscillator signal: signal-rate
- Current envelope position in milliseconds: floating point control-rate
- Output signal

Dependencies on other components

The [envelope~] abstraction provides main functionality; it is described below in section 5.3.9 . The [degrade~] object provides the amplitude quantisation.

Test notes

The amplitude envelope structure relies strongly on the [envelope~] abstraction, the structure around this is almost trivial. Most information regarding the testing can be found in section 5.3.9 .

5.3.8 Ringmodulation

Amplitude modulation (and/or ring modulation, see explanation) involves altering the amplitude of a signal by the output of another oscillator. That is to say the gain of one signal is altered in sympathy to the oscillations of another signal.

Strictly speaking, the structure implemented here is an *amplitude* modulation oscillator. The difference between *ring* modulation and *amplitude* modulation is that “ring modulation modulates two bipolar signals [having positive and negative values], while amplitude modulation modulates a bipolar signal with a unipolar signal [a signal having only positive values].” (Roads, 1996)

Both the amplitude and the “ring” modulation parameters are controlled by user interface elements (see section 5.5.3 on page 100) consisting of [function] objects. Since referring to both with the word “amplitude” could cause confusion, the modulation structure here is referred to as a “ring” modulator, even though this is strictly incorrect.

Options, Problems, and Decisions

Some of the options to consider when designing this component were the choice of waveshape for the modulating oscillator. As explained previously in section 3.2, the YM2149 was able to use arbitrary modulation waveforms by using software techniques to modulate the amplitude register of a channel, or by using the envelopes looping at audio rates. The Commodore SID chip was able to use the output of other channels as the modulation oscillator, enabling complex chains of ring modulated oscillators. The choice was made to use a square ring modulation oscillator waveshape, this is primarily due to the high number of sidebands that are produced as a result.

The use of step sequences to control the frequency and depth of the modulation was considered. However, the chosen alternative was an envelope control for the frequency of the oscillator.

The decision was made to allow the frequency of the modulating oscillator to act as an LFO, allowing stuttering or echo effects to be created when combined with the amplitude envelope. The phase of the ring modulation oscillator is optionally reset to zero every time a “note on” message is received. If the phase is not reset the oscillator free-wheels, potentially producing cross rhythms when triggered by “note on” messages with a rhythm of their own.

Specification

Provide amplitude modulation of an input signal by a square wave modulation oscillator.

The frequency of the modulation must be specifiable and alterable by means of an envelope.

The depth of the modulation must be specifiable and alterable and must not cause the output signal to exceed the limits of -1.0 or 1.0.

Final solution

The final modulation structure is presented below for convenience. The reader is strongly encouraged to view the full synthesis structure shown in Figure 5.5 on page 58 to review the context.

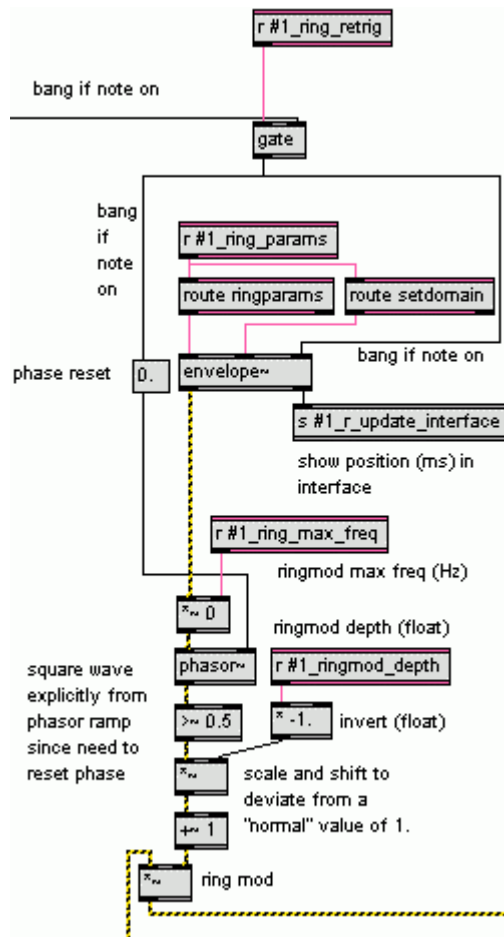


Figure 5.19 - Final amplitude modulation MaxMSP structure

The important features to note in Figure 5.19 are the phasor-derived square oscillator, the method by which the phase of the oscillator is reset, the method by which the [envelope~]

abstraction is used to set the frequency of the oscillator, and the method by which the depth of the modulation is set by scaling and shifting the oscillator output.

Firstly, a [phasor~] object is used to generate the square waveform due to the ability to reset the phase of the ramp to zero. The band-limited [rect~] does not have this capability. The [gate] object is used to filter the “note on” messages, optionally allowing them to pass to the [phasor~] object depending on the status of a toggle in the user interface, received by the topmost pink [r] object.

The frequency of the oscillator is set by multiplying the output of the [envelope~] abstraction by a value set in the user interface. The [envelope~] abstraction has a maximum value of 1.0. The value of the multiplying factor is received by another pink [r] object.

Finally, the scaling and shifting of the oscillator is performed to produce a minor variation in amplitude when the control is set low, and a full toggling of gain from 1.0 to 0.0 when the control is set to full. Figure 5.20 explains this graphically.

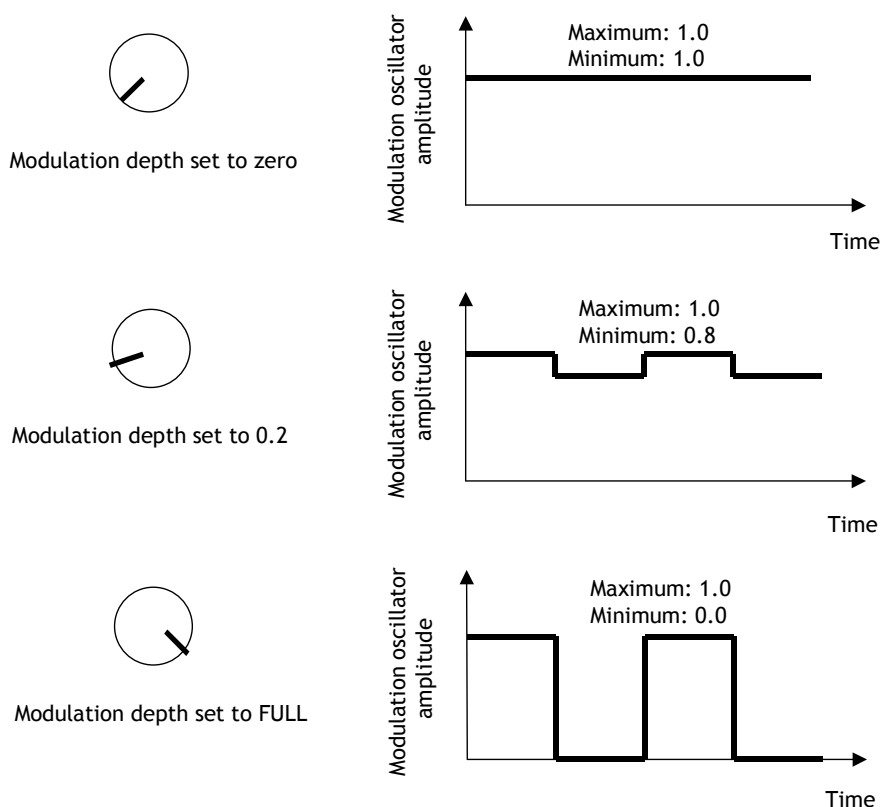


Figure 5.20 - Intended functionality of modulation depth control

Inputs

- Envelope parameters from user interface as [r] object:
 - “All points in line format” preceded by the string “envparams”
 - Envelope duration (control-rate integer) preceded by string “setdomain”
- “Note on” trigger: control-rate bang message
- Retrigger on “note on”: control-rate integer
- Maximum modulator frequency: control-rate floating point

- Modulation depth: control-rate floating point
- Input signal

Outputs

- Output signal

Dependencies on other components

The [envelope~] abstraction provides main functionality; it is described below in section 5.3.9 .

Test notes

The structure relies strongly on the [envelope~] abstraction, the structure around this is almost trivial. Most information regarding the testing can be found below in section 5.3.9 .

5.3.9 The [envelope~] abstraction

This abstraction is used to generate the control values for amplitude envelope and ring modulation oscillator frequency.

This abstraction is similar to the [clockedseq~] abstraction in that it mediates between the display of parameters in the user interface and the actual control signals taking place in the synthesis structure. Where the [clockedseq~] abstraction turns a graphical representation of a step sequence into an appropriate signal-rate control signal for the synthesis structure, the [envelope~] abstraction also complements a user interface component; in this case the UI component is a graphical representation of an envelope specified in terms of discrete x,y co-ordinates.

Specification

This component must accept data defined at the user interface in terms of a breakpoint [function] object describing an ramp sequence in terms of x,y co-ordinates.

The component must produce a signal-rate output that interpolates between the points.

The component must reproduce the envelope when a note trigger is received, without referring back to the original [function] object data. That is to say, the component must store a copy of the envelope points to be able to reproduce them again later.

The component must accept the envelope duration as a control-rate integer value specifying the length of the envelope in milliseconds, the scaling of the points in accordance with this duration is not dealt with by this component. This envelope duration value is merely used to produce a control-rate output estimating the current time position of the envelope for user feedback to the user interface.

Options, problems, and decisions

Some of the other sections discussing the options, problems, and decisions have touched on the idea of a central matrix to control the routing of all modulation signals as step sequences.

The possibility of “smoothing” the jumps in the step sequence with optional interpolation between steps was also considered. This might be useful for amplitude control where sudden jumps may produce undesirable clicks in the output. Having said that, replaying a deliberately stepped amplitude control sequence at audio rates would no doubt produce interesting sonic results.

During the experimental stages, a prototype of the user interface was made using a breakpoint [function] object to describe an envelope in the place of a step sequence. Using an [mslider] object to “draw” a smooth envelope is a difficult task since the user must specify each point in the smooth envelope curve exactly. Conversely, the use of a breakpoint function object enables the user to specify only the points that are important, the system interpolate all the steps in between.

The change from step sequence via an [mslider] in the user interface to an envelope via a breakpoint [function] object required a new method of transporting or converting the data specified at the user interface to a signal generator in the synthesis structure.

More information about this can be found in section 5.5 and 5.6 (pages 97 and 105), but essentially the problem boils down to mouse movements manipulating the data held inside the [function] object not being output by the [pattrstorage] object that [convertdata] and [transportdata] rely on for communicating changes at the user interface level to the synthesis structure.

The workaround for this problem involves a “bang” object that re-triggers the output of the envelope every time a point is changed with the mouse, and number of [s] and [r] objects embedded into the user interface, transmitting data from one of the outlets of the [function] object, rather than relying on the [pattrstorage] object. A side effect of this workaround is that manipulation of the envelope data with the mouse in the user interface causes retriggering of the envelope in the synthesis structure.

A related problem is that early versions of the system would not update the envelope when using MIDI zones to switch between different sound parameter data rapidly (in a 4/4 time signature at 120bpm, switches between different percussion sounds triggered at 16th note intervals would occur every 125ms).

This problem is related to the structure set up to transport data from the user interface, and the method used to store and retrieve parameter data (see the section on parameter storage). A brief overview of the problem is that when changing between two stored sounds using the [pattrstorage] object, the behaviour of the envelope [function] object in the user interface causes it to partially ignore the change. The graphics update, but the recalled values are not sent out. The result is that the synthesis structure never receives any new envelope information.

A workaround is to force the [function] object to output the data held within it every time the [pattrstorage] object receives a message to recall a stored sound by sending a “bang” message to the [function] envelope object, outputting the envelope data to the synthesis structure.

Another issue is the conversion of the data from the [function] object to and from symbols, the [tosymbol] and [fromsymbol] objects truncate floating point numbers, reducing the accuracy of low values in both the x and y axis.

The [tosymbol] and [fromsymbol] objects also feature a limitation of 2048 characters in the list message (Cycling'74, 2005) which can present problems with complicated envelopes. This limit may seem very high, but due to the floating point decimal values being transmitted as character strings, the number of characters in a list could reach this limit faster than initially expected.

Final solution

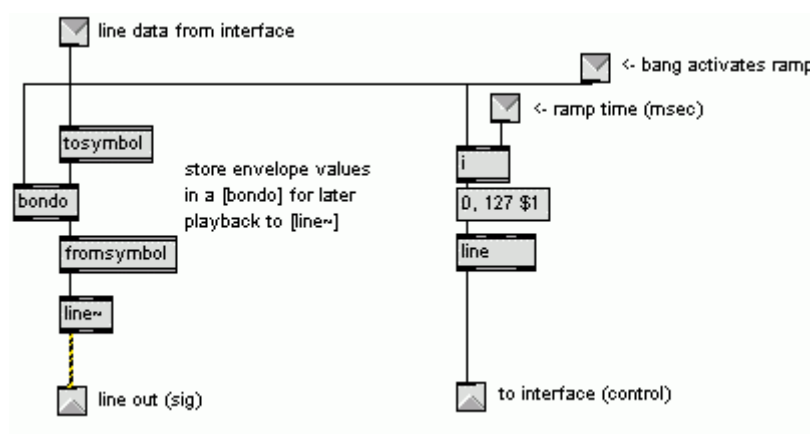


Figure 5.21 - Final envelope MaxMSP structure

The final solution involves using a [bondo] to store the breakpoint data, and a [line~] object to generate the signal-rate ramps.

Finally, a [line] object is used to generate a ramp that does not include any of the data from the breakpoint editor but simply lasts the same length of time as the envelope. This is used to indicate the time position of the signal-rate envelope generator for feedback to the user interface.

Inputs

- Line data from user interface: (from [function] 2nd outlet “all points in line format”)
- Ramp time in milliseconds: control-rate integer
- “note on” trigger: control-rate bang message

Outputs

- Output envelope: signal-rate
- Envelope position: floating point control-rate

Dependence on other components

The component depends on structures set up in the synthesis structure to filter data sent from the user interface, separating the breakpoint data from the duration of the envelope. It also depends on structures set up in the user interface to transport the data to the synthesis structure correctly, and on the correct data being sent from the [function] editor’s 2nd outlet in the first place.

The dependence on the [tosymbol] and [fromsymbol] objects revealed limitations in floating point decimal accuracy and in terms of the 2048 character limit.

Testing notes

The testing revealed the strange behaviour outlined in the “problems” section whereby the synthesis structure would not receive new envelopes when changing sounds.

A minimum value of two milliseconds is enforced at the user interface for the envelope duration. No maximum value is enforced.

5.4 Interface components

This section covers the graphical user interface (GUI) and machine interface (MIDI). As explained in section 5.2 on page 55, many revisions of the overall system were created, the decision to separate the interface from the synthesis structure certainly helped in terms of minimising the amount of development time devoted to restructuring the system after a change was made to either section.

The author’s lack of experience in user interface design presented a challenge in terms of designing an interface that is suitable for both live performance and in conjunction with a sequencer.

5.4.1 The [Interface] abstraction

This component collects together the various interface components that allow the user to interact with the system. MIDI processing is also performed in by this component, so the term “interface” applies to both user and machine interfaces.

Options, problems and Decisions

Early versions of the user interface were not contained in an abstraction, synthesis and MIDI zone parameters were accessed by separate windows. This had the potential to be very confusing for the user where several windows for different channels were open.

It is important to note that the appearance and layout of a music system may affect the way it is used. By combining all user interface areas into a single window, one for each channel; the presence of MIDI zone filtering is made more obvious and hopefully encourages its use.

Final solution

Compare Figure 5.2 on page 55, showing the overview of the system to the more detailed diagram below showing the organisation of the interface components within the overall structure.

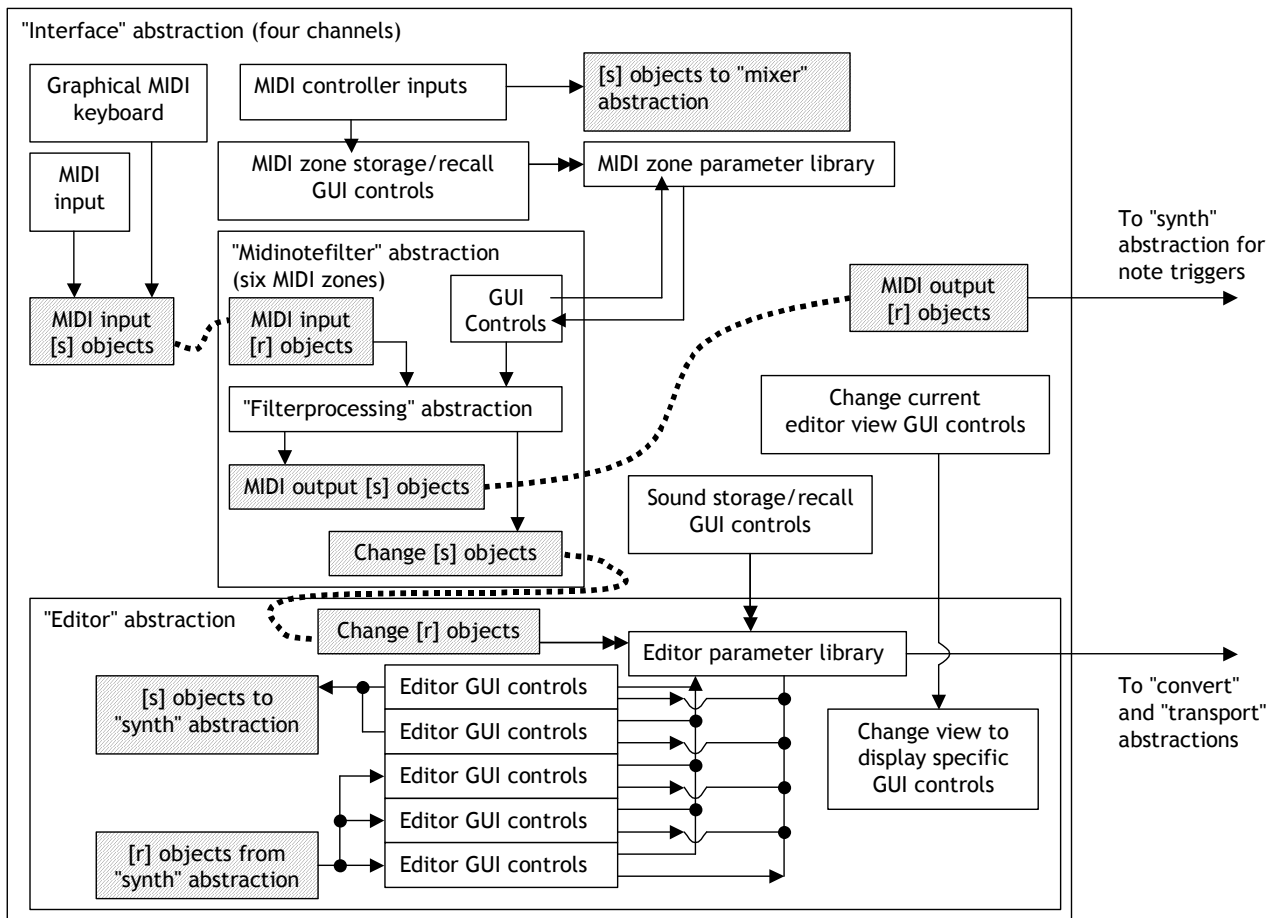


Figure 5.22 - Interface block diagram closely resembling MaxMSP structure

The interface structure makes heavy use of [s] and [r] objects in order to reduce the number of connecting control-rate cables between the abstractions. A single [s] object can be connected to multiple instances of an [r] object within abstractions such as the [midinotefilter] abstraction. To help increase readability, pairs of [s] and [r] objects are indicated in Figure 5.22 by a thick dotted line. Double-headed arrows in the diagram indicate the flow of instructions to recall parameter data from “library” storage.

Careful examination of Figure 5.22 will reveal that all MIDI data flows through the [midinotefilter] abstractions and that the abstractions also configure the “editor parameter library”. The consequence of this is that a user must configure at least one filter to pass a certain range of MIDI notes through to the synthesiser, and to recall a certain group of synthesis parameters before a note can ever be heard.

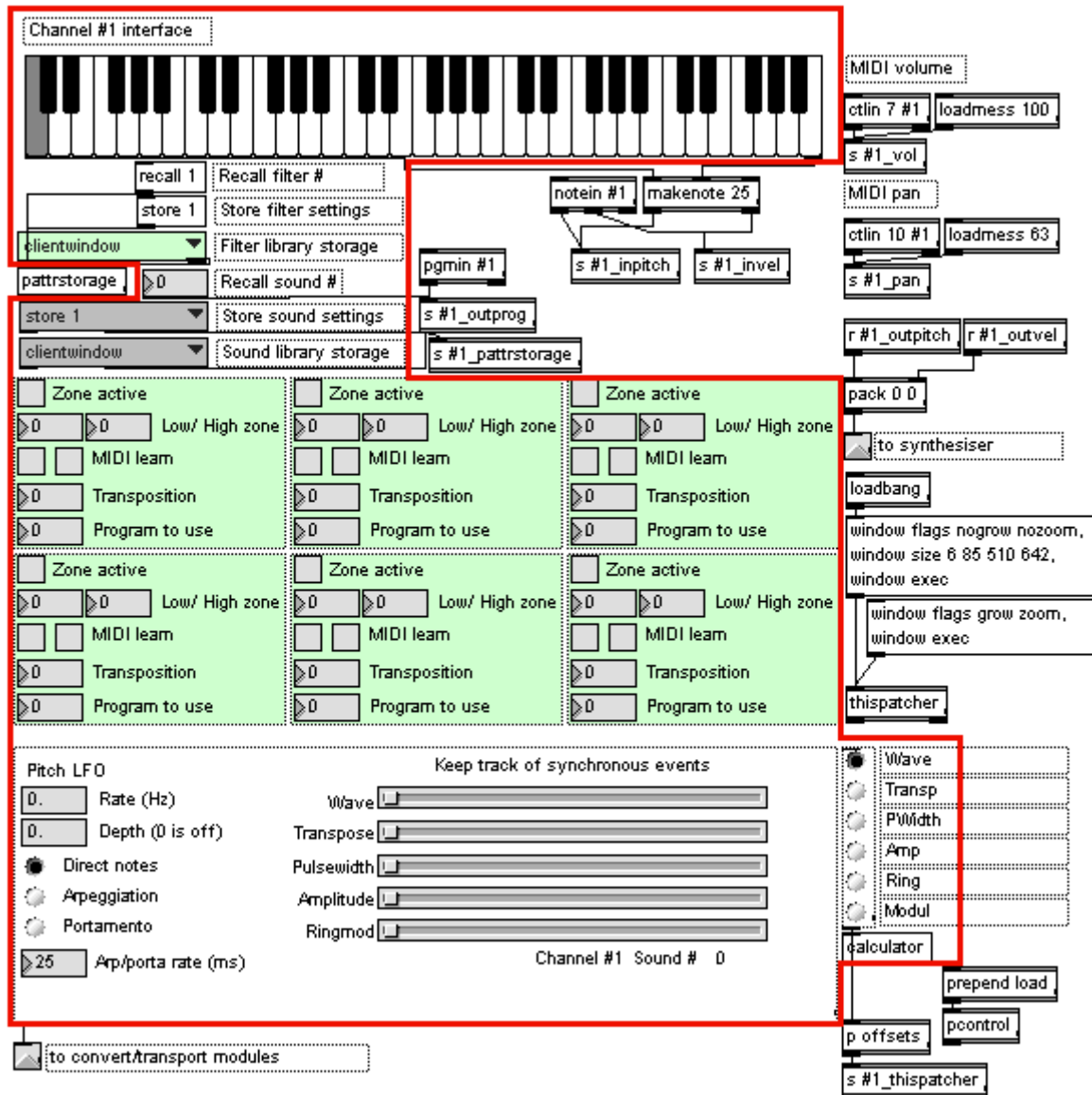


Figure 5.23 - Final interface abstraction MaxMSP structure

Figure 5.23 shows the final interface MaxMSP structure. The red border is not visible to the user but is included to identify the objects that the user will see when the patch is locked. Objects outside the red border provide functions that the user does not interact with. This convention is used throughout this section.

Six groups of synthesis parameters are available in the [editor] abstraction, but only one can be visible at any time. The strip of radio buttons in the lower right of Figure 5.23 control the visibility of the GUI controls by offsetting the viewing area of the GUI objects arranged within the [editor] abstraction.

The [thispatcher] object is used to prohibit the user from resizing the interface window. The structure involving the [pcontrol] object at the lower right is used to open the calculator window. The calculator is discussed in section 5.5.4 on page 101.

5.4.2 [midinotefilter] abstraction

The [midinotefilter] abstraction is part of the [interface] abstraction that processes MIDI data. It has a user interface area that is displayed by the [bpatcher] object in the [interface] abstraction. Six instances are called, operating in parallel.

Figure 5.24 shows that all MIDI data for each channel flows through the six [midinotefilter] abstractions in parallel. MIDI note triggers not matching the specified range in one of the abstractions is discarded but may still pass through to the synthesiser abstraction if the note is passed through by another of the parallel filters. If two zones have overlapping ranges, duplicate MIDI note triggers will be sent to the synthesiser.

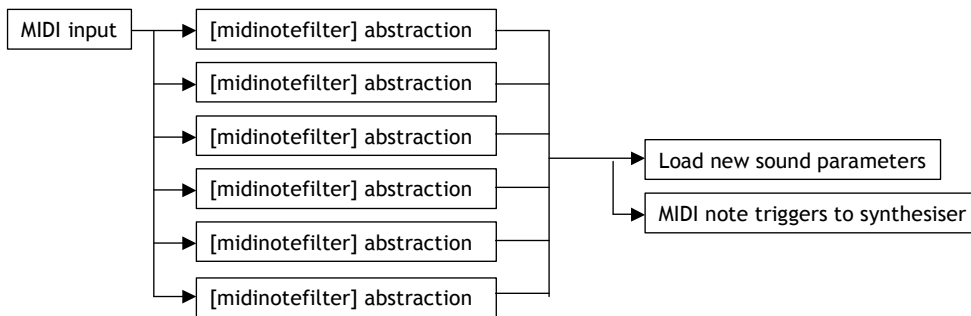


Figure 5.24 - Parallel processing of [midinotefilter] abstraction

The internal structure of the [midinotefilter] abstraction is shown below:

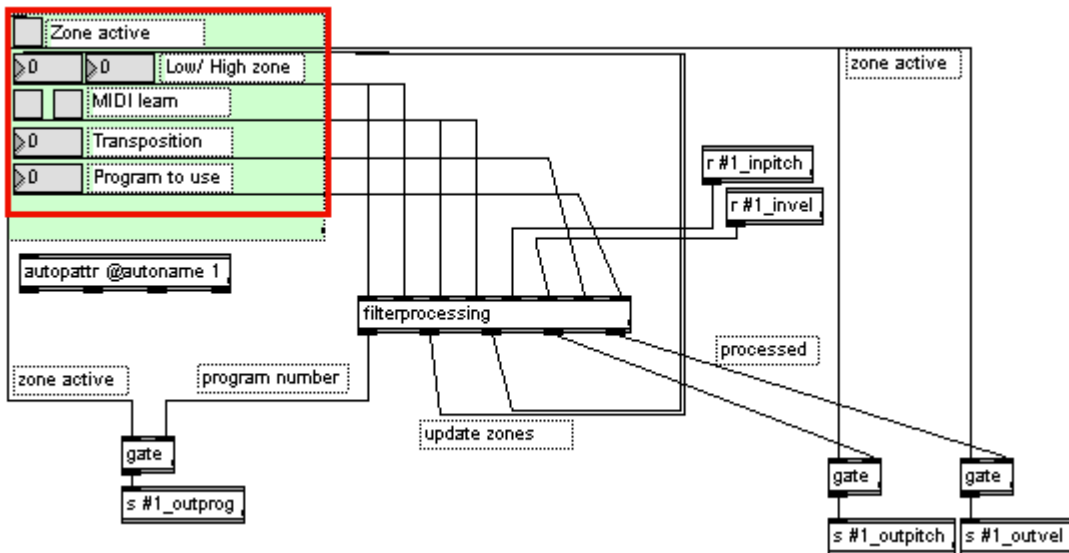


Figure 5.25 - Final note filter MaxMSP structure

Again, the red border indicates the objects that the user will see. The central object, both conceptually and figuratively is the [filterprocessing] abstraction outlined below. The gate objects allow the user to temporarily disable the zone by causing the gate objects to discard or pass the processed data. The [r] objects “#1_inpitch” and “#1_invel” are passed the MIDI note pitch and velocity for the current channel in parallel with other instances of the

abstraction. Similarly the [s] objects at the bottom of the figure pass control data back to the parent patch in order to select the appropriate sound, and pass through the MIDI note data.

Input

- MIDI note pitch: [r] object, control-rate integer
- MIDI note velocity: [r] object, control-rate integer

Output

- MIDI note pitch: [s] object, control-rate integer
- MIDI note velocity: [s] object, control-rate integer
- Sound program to recall: [s] object, control-rate integer

Dependence on other components

The [filterprocessing] abstraction handles the core functionality of this abstraction. This abstraction also depends on the [autopattr] object to communicate with the [pattrstorage] object in the parent patch. This enables the storage and recall of the all MIDI zones in an entire channel. Without this behaviour, the user could potentially be forced to remember $7 * 6 * 4 = 168$ parameters for the zone configuration of each channel.

Testing notes

The eventual destination of the number input objects in the user interface of this abstraction are the [filterprocessing] abstraction, described in the next section.

The low/high zone, input boxes are limited to valid MIDI note numbers (0-127). No checking is performed to see if the low zone value is greater than the high zone value (or vice versa), or if two instances of the abstraction have overlapping zones. When the zone limits are invalid (low value higher than high value or high value lower than low value) incoming MIDI data is simply discarded due to the logical statements in the [filterprocessing] abstraction.

The transposition input box is limited to a range of -127 to +127. No checking is performed to see if the transposition of notes will result in an invalid MIDI note number, for example by shifting MIDI note 87 up by 127 to form an invalid MIDI note pitch of 214. The eventual destination of this MIDI note pitch is the [mtof] object in the [synth] abstraction which handles invalid MIDI pitches by extending the upper and lower ranges as if the MIDI pitches were in fact valid. A strictly invalid note pitch of 214 produces a frequency twice as high as the valid MIDI pitch 107.

The program input box is limited to a range between 0 and 16 (0 being a temporary storage space used by the [pattrstorage] object). To prevent the user inadvertently making changes to the input box and recalling old parameter data without first saving the data currently being edited, the [filterprocessing] abstraction only allows the recall of parameter data if there is an incoming note message.

5.4.3 [filterprocessing] abstraction

This component is used by the [midinotefilter] abstraction to detect the presence of MIDI note pitches falling within a specified range or “zone”. It uses logical operators and gating of control-rate messages to provide this functionality. A flowchart of the basic logic is presented below. The functionality enabling “MIDI learn” (automatic zone definition based on MIDI input rather than keyboard or mouse input) is omitted for simplicity.

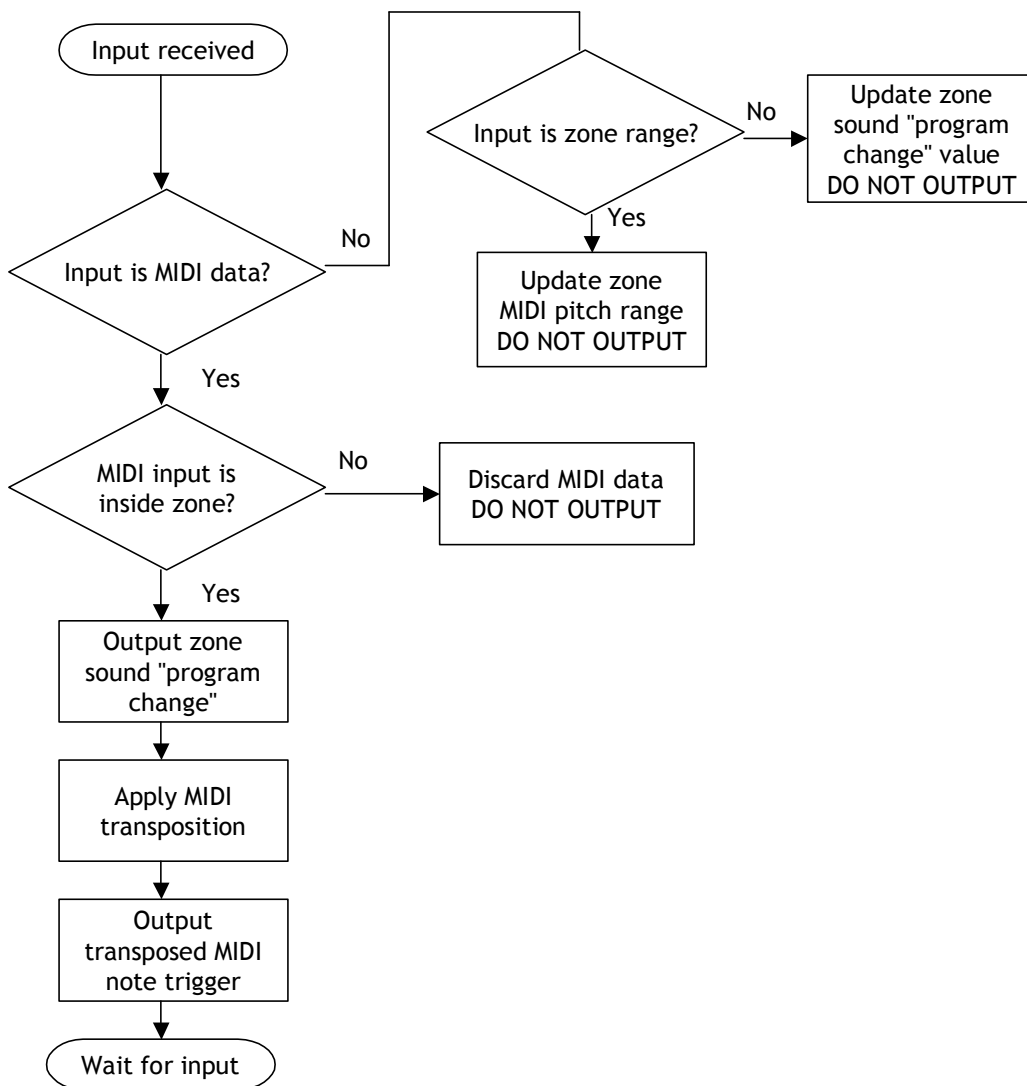


Figure 5.26 - Flow chart of [filterprocessing] logic

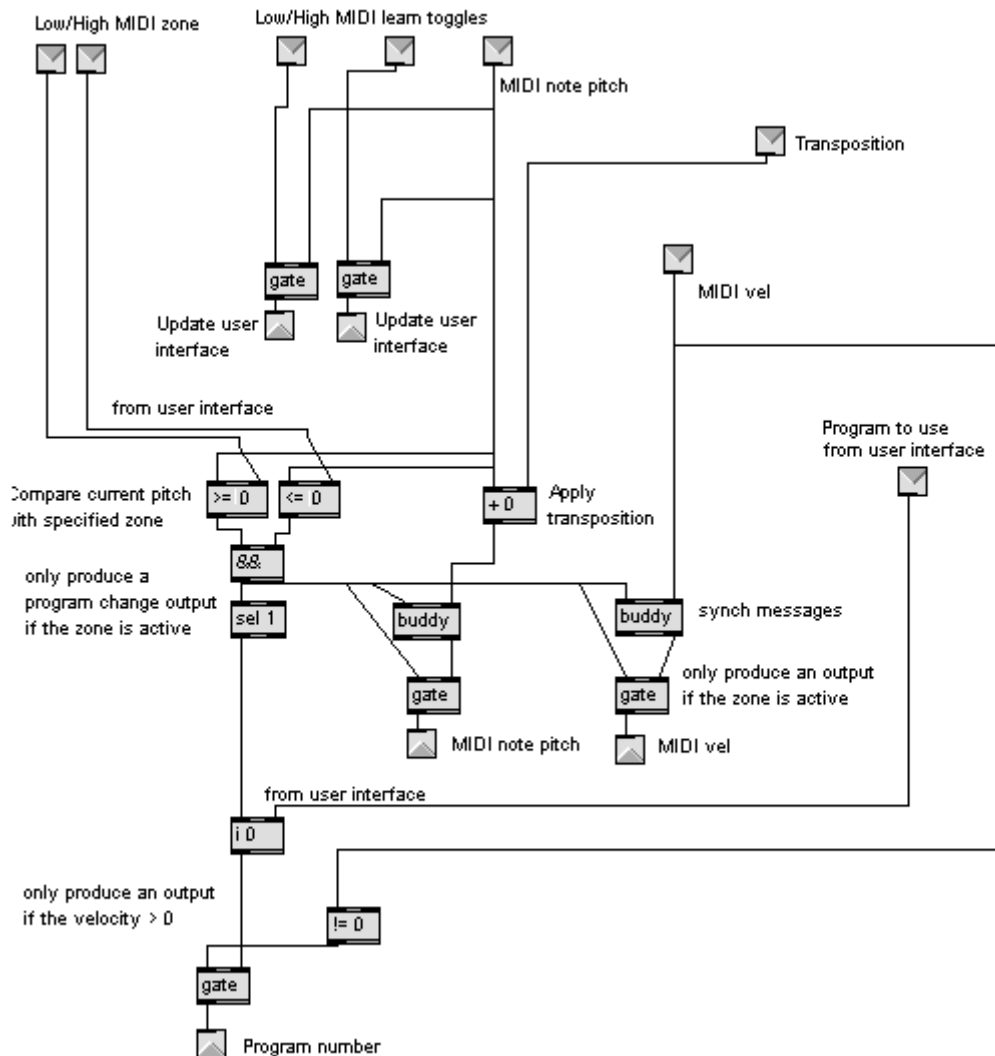


Figure 5.27 - Final [filterprocessing] MaxMSP structure

Final solution

Careful examination of Figure 5.27 will reveal the use of [buddy] before [gate] objects near the centre of the diagram. The reason for these objects is to counteract the problem of processing delays.

During testing of the MIDI zone processing elements, a problem often encountered was the intermittent behaviour of the zone processor. Sometimes MIDI data would be passed through, other times not. The problem was eventually narrowed down to the [gate] objects by temporarily placing [bang] objects in the user interface area to flag various conditions. The group of logical operators would send the signal to open the gates slightly after the MIDI messages that to be gated had already been blocked.

The [buddy] object retains the MIDI message until the logical operators have finished processing. After this, the [buddy] releases the MIDI message and the gate open/close signal simultaneously, ensuring that the [gate] will never “miss” a MIDI message due to timing problems. The drawback is that this adds a processing delay to all MIDI messages, but this delay is negligible and a necessary trade-off in order to ensure accurate MIDI zone handling.

To prevent the user inadvertently making changes to the “program number to use” input box in the user interface of the [midinotefilter] abstraction, causing the [pattrstorage] object to recalling old synthesis parameter data without first saving the data currently being edited, the [filterstorage] abstraction only outputs a recall message if there is an incoming note message.

This is accomplished by using an [i] object to store the program number coming from the user interface, only outputting this number if the note-on message matches the zone ranges AND a the velocity of the note is still greater than zero. The author found during testing that omitting the velocity check causes any change in the “program to use” input box in the user interface to be echoed if the last note to be checked fell within the zone. The changes are echoed even if the note is now “off”, unless the velocity check is also included.

Inputs

- MIDI note pitch: control-rate integer
- MIDI note velocity: control-rate integer
- Low zone limit: control-rate integer
- High zone limit: control-rate integer
- MIDI learn toggles: control-rate integer
- MIDI transposition: control-rate integer
- Program change to issue: control-rate integer

Outputs

- MIDI note pitch: control-rate integer
- MIDI note velocity: control-rate integer
- Low zone limit to interface: control-rate integer
- High zone limit to interface: control-rate integer
- Program change: control-rate integer

Dependence on other components

This component depends on several standard objects including logical operators such as [<=], [>=], [&&], [!=], along with other objects such as [buddy], and [gate].

Testing notes

The limits of the low/high zone ranges, program to use, and transposition input boxes are described in the [midinotefilter] abstraction section 5.4.2 , on page 91.

5.5 [editor] GUI abstraction

The [editor] GUI is part of the [interface] abstraction. A small part of the GUI is displayed by a [bpatcher] object embedded in the [interface] abstraction. The MaxMSP structure showing all editor parameters can be found in the appendices.

The reader is encouraged to refer back to Figure 5.4 and Figure 5.5 on pages 57 and 58 respectively to make sure the synthesis structure is firmly understood.

Each section of the synthesis structure is controlled and configured by elements of the [editor] structure, a firm understanding of the [synth~] abstraction will clarify the pairing

between the source of the specified values at the [editor] and eventual destination of the [synth~] abstraction.

The [editor] abstraction is strongly linked to the [convertdata] and [transportdata] abstractions. Most of the objects and structures in the [editor] abstraction rely on these two abstractions to communicate their values to the [synth~] abstraction. The reader is strongly encouraged to read the information on parameter storage in section 5.6 in parallel to this section.

Options, problems and decisions

Some objects send data to the synthesis structure by way of the editor parameter library and the [convertdata] and [transportdata] abstractions. Others use [s] and [r] objects to communicate with the synthesis structure directly. The individual decisions to use one method over another are discussed in the relevant section

Early versions of the GUI involved separate windows for separate synthesis parameters; one version involved a long scrolling window presenting all parameters in one window, but required a high resolution display to view all parameters at once.

The final solution takes inspiration from the MLine editor (Figure 2.8 on page 33). A strip of radio buttons in the lower right of Figure 5.23 (page 90) control the visibility of the GUI controls by offsetting the viewing area of the GUI objects arranged within the [editor] abstraction. The [p offsets] object in the very lower right of Figure 5.23 contains a sub-patch structure that converts the numeric output of the radio buttons to [bpatcher] offset locations.

5.5.1 Control methods

Although the editor allows the user to control a wide range of synthesis parameters, the methods of control can be grouped into three areas:

1. Step sequences
2. Single inputs (including switches, numerical entry boxes, and buttons)
3. Envelopes

Table 5.1 below summarises the synthesis parameters and the control methods used in the [editor] user interface abstraction.

Synthesis parameter	User interface control method
Active waveform	Step-sequence & Single inputs
Pitch Transposition	Step-sequence & Single inputs
Square/Pulse duty cycle	Step-sequence & Single inputs
Ring modulation	Envelope & Single inputs
Amplitude	Envelope & Single inputs
Pitch LFO	Single inputs
Arpeggiation/portamento	Single inputs

Table 5.1 - Summary of control methods used in [editor] abstraction

The flow of data as outlined in Table 5.1 is presented in below. This diagram should help clarify the pairing of information specified at the user interface, and the parameters being

controlled in the synthesis structure. The middle strip uses abbreviations (C) and (T) to indicate which of the abstractions [convertdata] and [transportdata] are used to communicate with the synthesis structure.

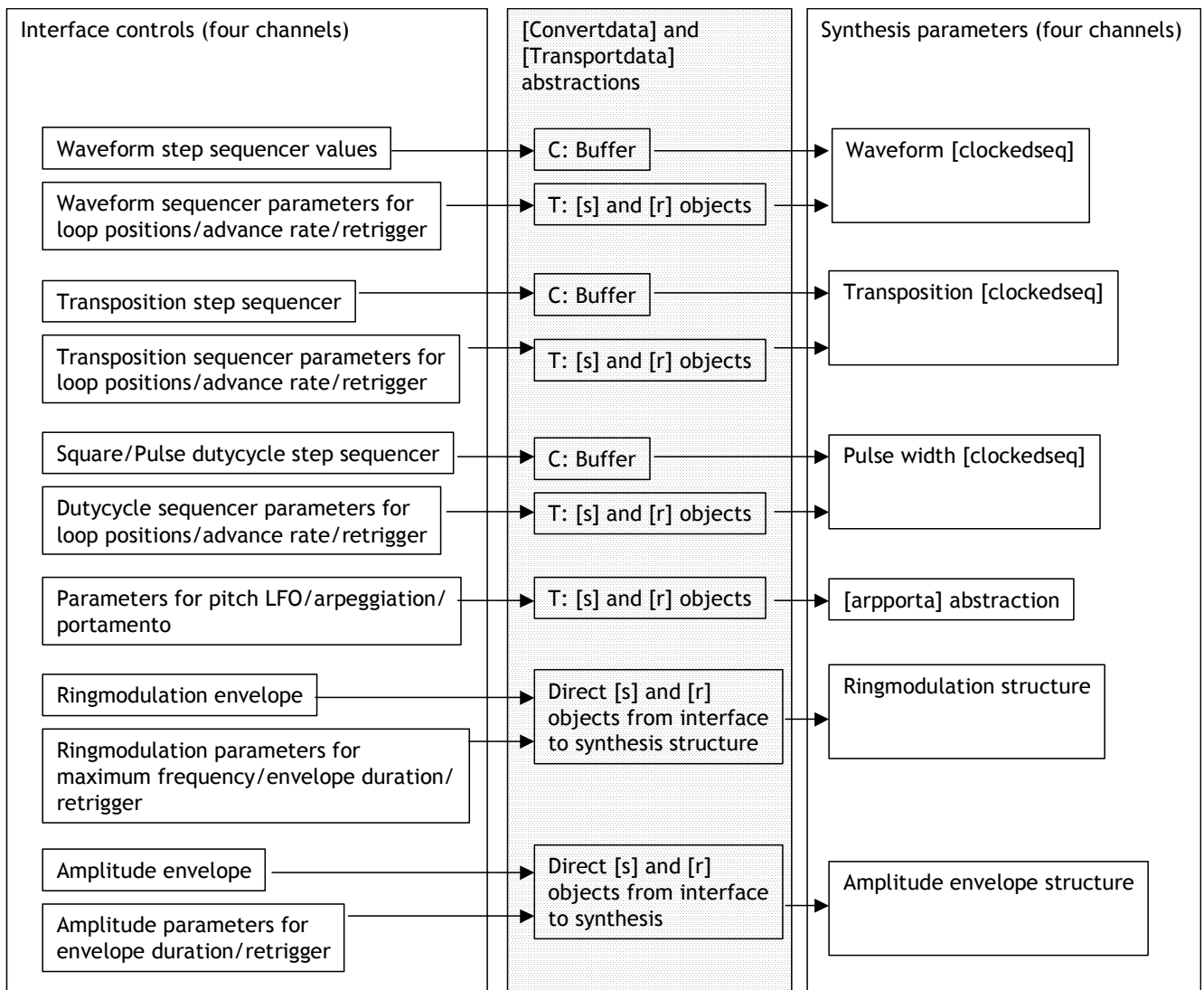


Figure 5.28 - Pairing of interface controls with synthesis parameters

5.5.2 Step sequencers

This section describes the design and indented use of the step sequence control structure used in the [editor] user interface abstraction. The discussion uses screenshots from the transposition area of the [editor] however the principles of its operation remain the same where the structure is used to control other parameters as outlined in Table 5.1.

The step sequencer takes inspiration from YMTracker as shown in Figure 2.5 on page 30.

Options, problems, and decisions

Early prototype versions of the step-sequencer used a row of number input boxes to set the values; later versions used the [mslider] object which represents values as vertical or horizontal stacks of sliders contained within a bounding box. This object was chosen for many reasons:

- It allows the user to drag the mouse over the graphical area, setting a large number of sliders under the mouse pointer to the appropriate values in one motion without needing to click on each slider individually.
- Should the decision be made to allow the user to specify the size of step sequences, the object scales the number of pixels each slider occupies appropriately within the graphical area.

Drawbacks with using this object are that the vertical resolution of the sliders is limited to the graphical representation on the screen. If the slider output ranges from 0-999 but the graphical object is only 64 pixels high, the slider's output will not allow the full range of 1000 values to be accessed. Setting the slider to certain numbered values will be impossible due to the lack of 1000 unique addressable pixel positions for the mouse pointer to take.

To remedy this situation, a fine-tuning structure was created that allows the user to set the general shape of the desired sequence by using the mouse, and to set the value of certain sliders in the array to a specific value by using numerical input boxes.

Another drawback is that the object only allows control-rate outputs. If the step sequence is to be replayed at audio rates, a conversion must take place between the values specified in the [mslider] to a form that can be replayed at audio rates with signal-rate objects.

Another factor to be considered is the problem users may face when setting the loop positions for individual step sequences. With a large sequence, the absence of an indicator to show the current index position next to the step sequence display may cause the user to be unable to tell which values are being looped.

The solution is to provide an indicator that moves across the sequence, allowing the user to visually determine which position is currently being output, and therefore if the loop positions are correct.

To help users create sounds with synchronised changes multiple synthesis parameters in a step-sequence manner, these position index displays are duplicated and grouped together in another area of the [editor]. This is especially useful given the decision (discussed in section 5.3.1) to allow the user to specify different advance rates for each sequence. This opens up the potential for cross-rhythms, but the decision to display only a small part of the GUI at one time presents problems for users wishing to view parameter sequences simultaneously. By grouping the index positions into one simultaneous view, users can confirm that the sequences are advancing in the desired way without having to frantically switch between GUI views.

Re-ordering of step sequence data

In the original project proposal, the use of “drag and drop sequence re-ordering” was mentioned. This feature was not implemented, but steps toward it were made. A brief explanation of the intended functionality is as follows:

A user defines a looping step sequence, but decides that the order of the sequence is not quite right.

An abstraction was constructed that allowed the user to move all values in the sequence left or right by one position. Values at the “edges” of the sequence are not discarded, but rather

re-appear at the opposite edge. The intention was that by repeated use of this function, the user could rotate the sequence through different permutations, combining this with different sequence lengths and advance rates for different parameters could produce a vast variety of possible sounds.

Unfortunately, the development of the step sequence re-ordering abstraction was abandoned for several reasons:

1. The original objective on the project proposal was to provide a drag-and-drop re-ordering of step sequences.
2. More pressing matters (such as testing and documentation) took precedence over development in the later stages of the project.
3. The abstraction suffered from some serious flaws (most likely due to the experimental prototype nature of the implementation): sometimes losing data, sometimes crashing the MaxMSP software if certain initial conditions were not met.

Final solution

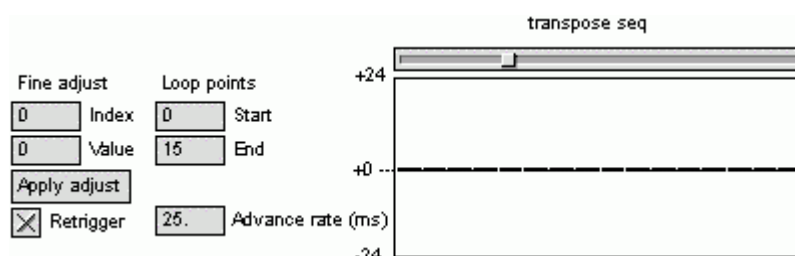


Figure 5.29 - Final step sequence MaxMSP structure

Figure 5.29 shows the final step sequence structure, the connecting cables have been hidden. The full [editor] abstraction, with connecting cables shown, can be found at the end of this section.

Differences between versions

As identified in Table 5.1, the sequencer structure is used in different contexts:

The transposition step sequencer uses positive and negative integers to specify positive and negative transpositions in semitone intervals.

The waveform sequencer uses only positive integers where different values represent different waveforms.

The duty cycle sequencer uses positive floating-point numbers to specify the width of the square/pulse oscillator.

Dependence on other components

The component depends on the [mslider] object to provide the graphical representation of the step sequence. The [fineadjust] abstraction provides accurate assignment of values to individual steps where the screen resolution and the size of the sequencer prevent accurate values being set with the mouse alone.

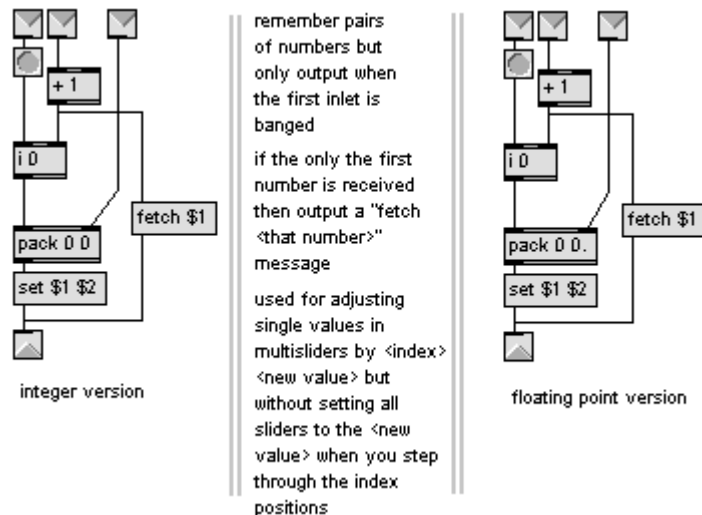


Figure 5.30 - Final fineadjust MaxMSP structure (both versions)

Two versions of this [fineadjust] abstraction were made, one dealing with floating-point numbers (for use with the pulse width/duty cycle sequencer) and one dealing with integers for use elsewhere. Figure 5.30 shows the minor difference is a floating point or integer argument in the [pack] object.

Testing notes

The eventual destination of the values specified in the user interface is the [clockedseq~] abstraction. The testing of this is outlined in section 5.3.1 beginning on page 60.

5.5.3 Envelopes

This section describes the design and indented use of the graphical envelope control structure used in the [editor] user interface abstraction. The discussion uses screenshots from the ringmodulation area of the [editor] however the principles of its operation remain the same where the structure is used to control other parameters as outlined in Table 5.1 on page 96.

Options, problems, and decisions

As explained in sections 5.3.7 , 5.3.8 , and 5.3.9 this choice to use an envelope to control the amplitude and ringmodulation synthesis parameters was made after the idea of step sequences being played at audio-rates was abandoned.

Had this decision not been made, the user may have been able to specify long wavetables with envelope shapes in them, gradually decaying to zero. Loop points could be specified that cause the last element in the wavetable to be replayed over and over (sustaining the final zero value) until the wavetable/envelope is retriggered.

This structure could be extended to provide ringmodulation by specifying a single period of a waveshape inside the wavetable. This time the loop points could be configured to cause the entire wavetable to repeat creating a modulation oscillator that causes ringmodulation.

Using a wavetable to specify the amplitude envelope would also enable synchronous amplitude scaling of specific waveforms if the clock speeds of the waveform and amplitude sequences were identical. With the chosen envelope/breakpoint function method, it is

difficult to create sudden steps in amplitude. The construction of complex amplitude alterations in sympathy with changes to other parameters controlled with step sequences may involve many mouse clicks.

Final solution

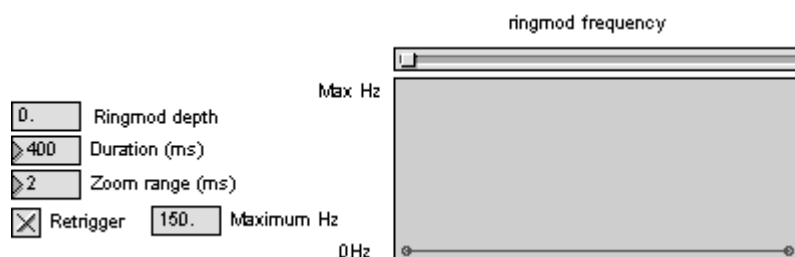


Figure 5.31 - Final graphical envelope MaxMSP structure

Figure 5.31 shows the final structure, the connecting cables have been hidden. The full [editor] abstraction, with connecting cables shown, can be found in the appendices.

Dependence on other components

This component depends on the [function] object to provide the graphical representation of the envelope.

Testing notes

The eventual destination of the values specified in the user interface is the ringmodulation and amplitude envelope structures that use the [envelope-] abstraction. The testing of the relevant areas are outlined in sections 5.3.7 , 5.3.8 , and 5.3.9 beginning on pages 80, 82, and 85 respectively.

5.5.4 [calculator] patch

This component is included to help users deal with a situation where two step sequences for two parameters (e.g. transposition, and waveform) each have a different number of values between the "loop start" and "loop end" positions, and need to be replayed at different speeds.

If a user wishes to specify two sequences of differing loop lengths which should advance at different interval rates, and that they also must reach their "loop start" and "loop end" points in synchronism; the user must calculate the ratio between the number of steps and the advance intervals of each sequence.

The calculator abstraction includes a simple interface whereby the user enters two sequence lengths and one advance rate; whereupon the calculator provides the advance rate for the other sequence.

Options, problems, and decisions

An alternative was considered where the user specifies two sequences of the same length, but wishes one sequence to advance more quickly than the other.

Another calculator could be created to allow the user to enter two differing advance rates and one sequence length; the calculator would then provide the appropriate length for the

other sequence to ensure that the "loop start" and "loop end" points are reached in synchronism.

Final solution

Here, a simple structure calculates the number of milliseconds elapsed after the number of wavetable steps for sequence A, and divides by the length of the steps in wavetable B. Colour coding helps the user identify which sequence length refers to which advance rate.

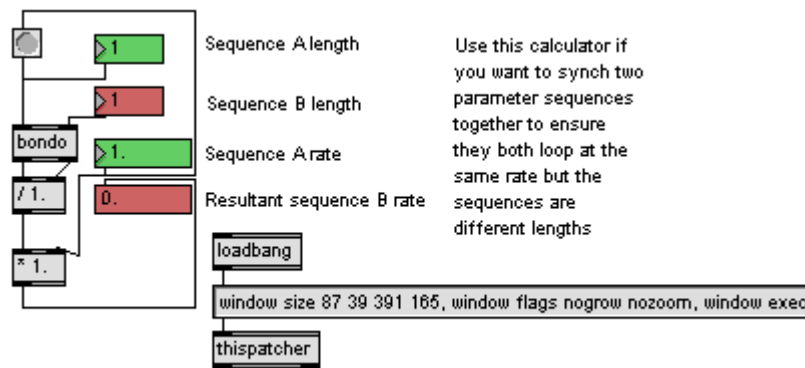


Figure 5.32 - Final calculator MaxMSP structure

The [bondo] object ensures that the user entering data into any input box causes the calculation to take place.

5.5.5 [mixer~] patch

This component is part of the main patch. It takes the monophonic output of the four [synth~] abstractions and allows the user to adjust the amplitude and stereo pan position of each channel.

A block diagram of the mixer abstraction is below:

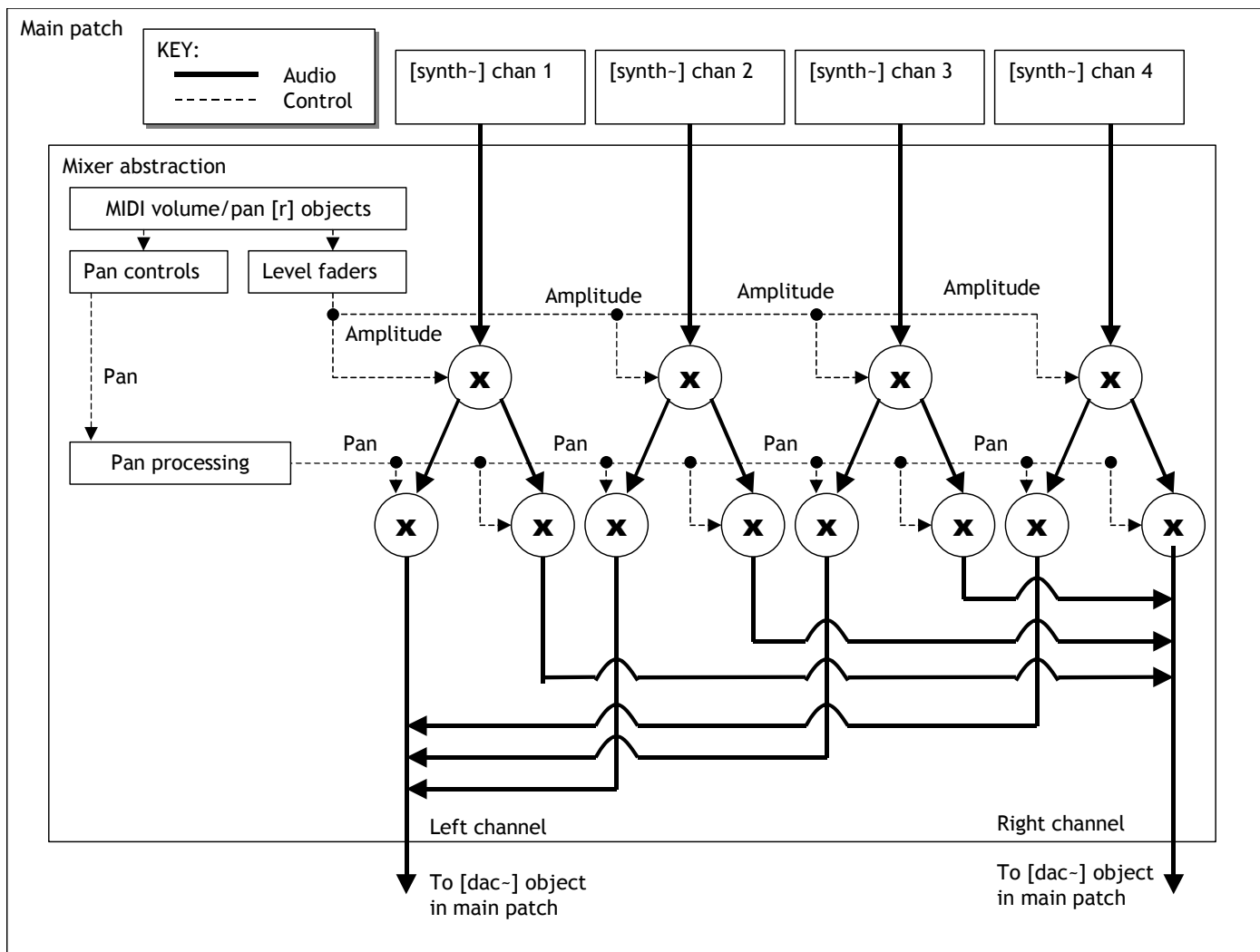


Figure 5.33 - Block diagram of mixer abstraction

The diagram above looks complicated, but the confusion comes from one structure being repeated four times with overlapping cables. All the diagram shows is that the individual mono signals are scaled by a factor determined by the level fader, and split into two signals which are then individually scaled by a value set by the pan control.

Options, problems, and decisions

The [mixer-] abstraction takes MIDI controller messages for volume and pan, allowing the user to adjust the amplitude and pan position

Choosing an appropriate mathematical manipulation for stereo pan was given careful thought. The Audio Processing course notes (Creasey, 2005) explain that a "linear pan law" provides mono compatibility, but suffers from a slight dip in perceived amplitude when the pan control is in the centre. Figure 5.34 below explains this graphically.

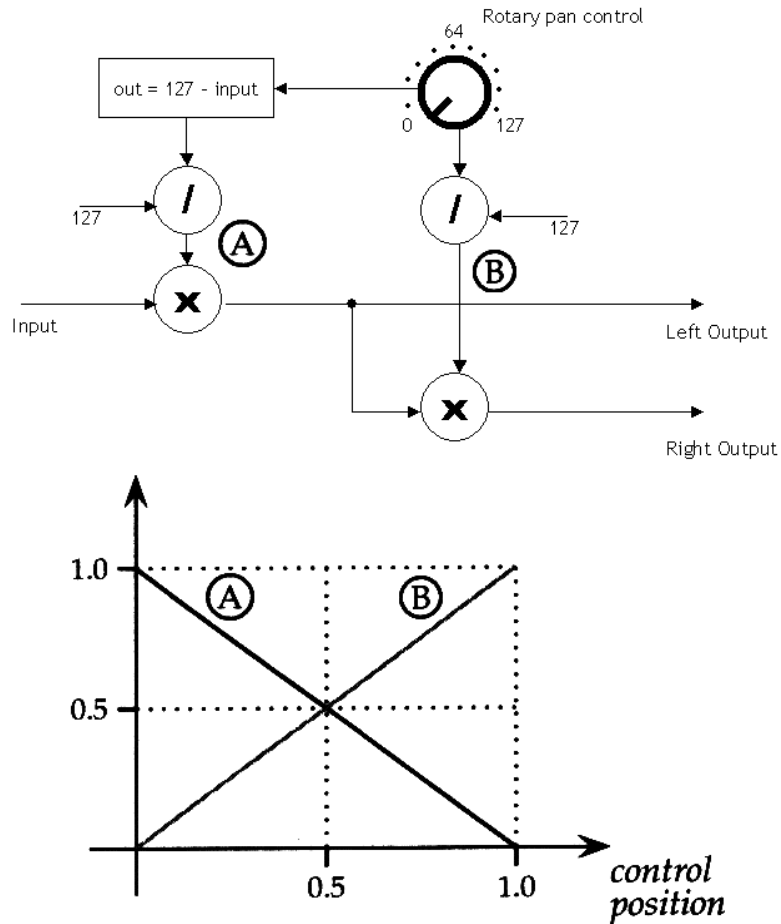


Figure 5.34 - Linear pan control
 Partially based on diagrams by Creasey (2005)

Several alternatives are possible including a logarithmic pan, or a "balance" control where the level of the left output stays constant as the right decreases. The final solution uses a linear pan control, adapted from the Audio Processing course notes.

Final MaxMSP structure

The final MaxMSP structure is shown below; again the red border is included to show the portion of the interface that is visible to the user.

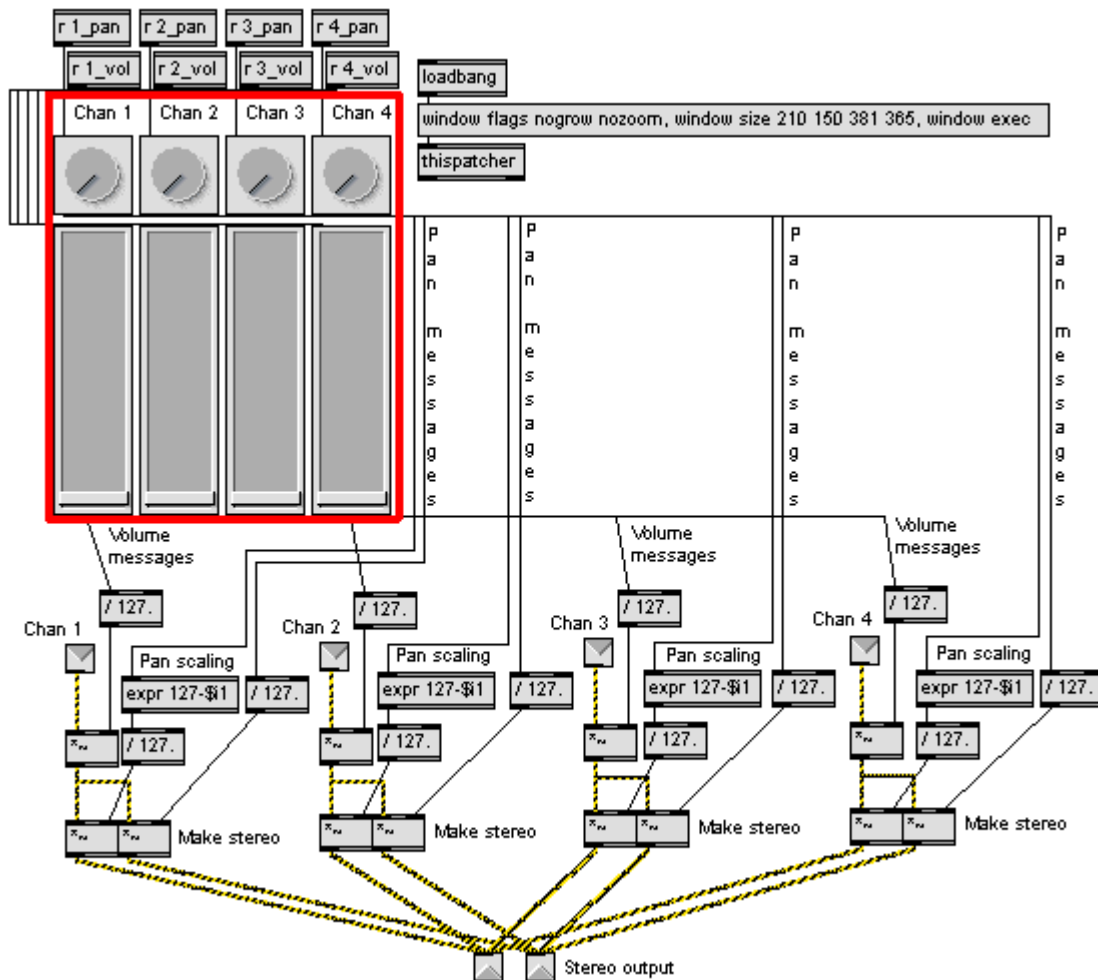


Figure 5.35 - Final mixer MaxMSP structure

As before, the red border indicates the area that is viewable by the user. Each channel has a rotary pan control and an amplitude fader. The values of these interface objects are altered by new values arriving from the [r] objects connected to MIDI controller input objects elsewhere in the [interface] abstraction.

The [loadbang] and [thispatcher] object are arranged to configure the mixer window as a fixed size with no scrollbars and a single close gadget. This method is also used in the [interface] abstraction.

5.6 Communication and storage of parameters

This section covers conversion of user interface data to synthesis parameters, including the storage and recall of data to and from disk. The section is presented here, toward the end of the implementation chapter, in order to be sure that the reader is comfortable with the ideas presented and the reasons behind the design decisions explained earlier in the report.

The reader is advised to review the following areas before this section:

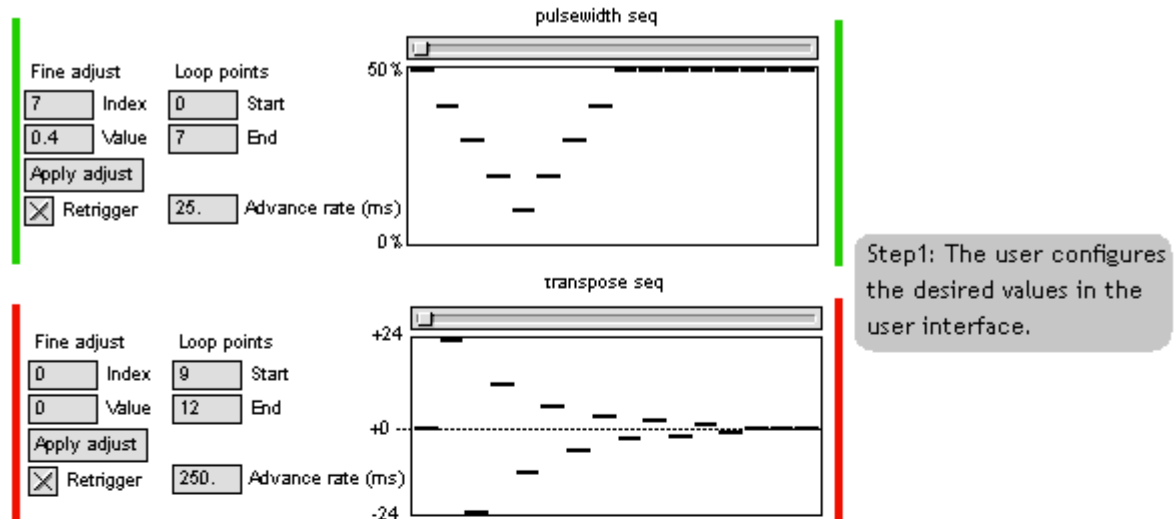
- The reader is strongly encouraged to refer to Figure 5.2 in section 5.2 (page 55) in order to be sure that the overall structure of the system is firmly understood.
- The structure of the interface is covered in more detail in Figure 5.22 on page 89. The reader is encouraged to study this image in order to be sure that the way the information moves around the interface structure, and how parameter data is stored and recalled. The “library” blocks in these diagrams (Figure 5.2, page 55 and Figure 5.22, page 89) where double-headed arrows in these diagrams indicate the flow of instructions to recall parameter data from “library” storage areas implemented as [pattrstorage] objects.
- Another image that may be useful to study before reading this section is Figure 5.28 on page 97, which shows the pairing of user interface areas with synthesis parameters. The [convertdata] and [transportdata] abstractions effectively form connections between the two sides of this diagram. The conversion of data held in control-rate user interface objects to a form that can be accessed by signal-rate objects in the synthesis structure is a step forward in the direction of the matrix controlled modulation routing system described in section 5.3.1 .

The [convertdata] and [transportdata] abstractions represent the “interface mapping” component in Figure 1.5 on page 9. As discussed, the inclusion of this interface mapping stage helped reduce the amount of development time devoted to restructuring after a change was made in either the synthesis or interface structure.

The storage of parameters relies on the use of two objects: [pattrstorage] and [autopattr]. The MaxMSP reference manual (Cycling'74, 2005) describes [autopattr] as an object to “manage multiple objects at once or expose them to pattrstorage”, and [pattrstorage] as “preset storage and general management for pattr objects, save and recall of pattr data”.

These objects are configured to keep track of user interface controls within the [editor] abstraction. By naming interface objects (for example “t_mslider” to indicate the transposition [mslider] object) any changes the user makes to named objects with the mouse or keyboard are output from one of the [pattrstorage] outlets as list messages with the name of the object preceding the new value.

Figure 5.36 shows the [pattrstorage] “clientwindow” view, listing all the named user interface objects it can “see” and their values. The green and red colour codes are included to help the reader identify the pairing between graphical and numerical representations of the same data.



Step1: The user configures the desired values in the user interface.

name	priority	interp	data
arporta_rate	0	lin.	25
bit_depth	0	lin.	16
env_duration	0	lin.	400
env_retrig	0	lin.	1
envelope_amp	0	lin.	400. 0. 1. 0. 0.507463 0 9.056231 1. 0 42.055931 0.492537 0 398.470673 0. 0
envelope_ring	0	lin.	400. 0. 1. 0. 0. 0 398.470551 0. 0
lfo_depth	0	lin.	0.
lfo_rate	0	lin.	0.
note_mode	0	lin.	0
p_clk	0	lin.	25.
p_mslider	0	lin.	0.5 0.4 0.3 0.2 0.1 0.2 0.3 0.4 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
p_retrig	0	lin.	1
pw_lpe	0	lin.	7
pw_lps	0	lin.	0
ring_duration	0	lin.	400
ring_max_freq	0	lin.	150.
ring_retrig	0	lin.	1
ringmod_depth	0	lin.	0.
t_clk	0	lin.	250.
t_mslider	0	lin.	0 24 -24 12 -12 6 -6 3 -3 2 -2 1 -1 0 0 0
t_retrig	0	lin.	1
ts_lpe	0	lin.	12
ts_lps	0	lin.	9
w_clk	0	lin.	25.
w_mslider	0	lin.	2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
w_retrig	0	lin.	1
ws_lpe	0	lin.	0
ws_lps	0	lin.	0

Step2: The values are stored and communicated to the [convertdata] and [transportdata] abstractions.

Figure 5.36 - Pattrstorage of interface objects

In order for these values to be communicated to the synthesis structure, the [pattrstorage] object is connected to the [convertdata] and [transportdata] abstractions. The [route] object is used to filter messages from the [pattrstorage] object according to the names of the objects (for example “t_mslider” again).

Referring back to Figure 5.2 on page 55, and Figure 5.3 on page 57: the [interface] abstraction can be seen with a channel number passed as an argument, the same argument is passed to the [convertdata], [transportdata], and [synth~] abstractions. By using the “#1” token to extract the arguments passed to each abstraction call, multiple instances of the [convertdata], [transportdata], and [synth~] abstractions can share the same structure but reference completely different values.

5.6.1 [transportdata] abstraction

Section 5.5.1 , beginning on page 96, summarises the use of the [transportdata] abstraction to communicate single values to the synthesis structure. Figure 5.37 below shows the internal structure of the [transportdata] abstraction, the “#1” tokens in the [s] objects ensure that the correct pink [r] objects in the [synth-] abstraction receive the messages for the correct channel.

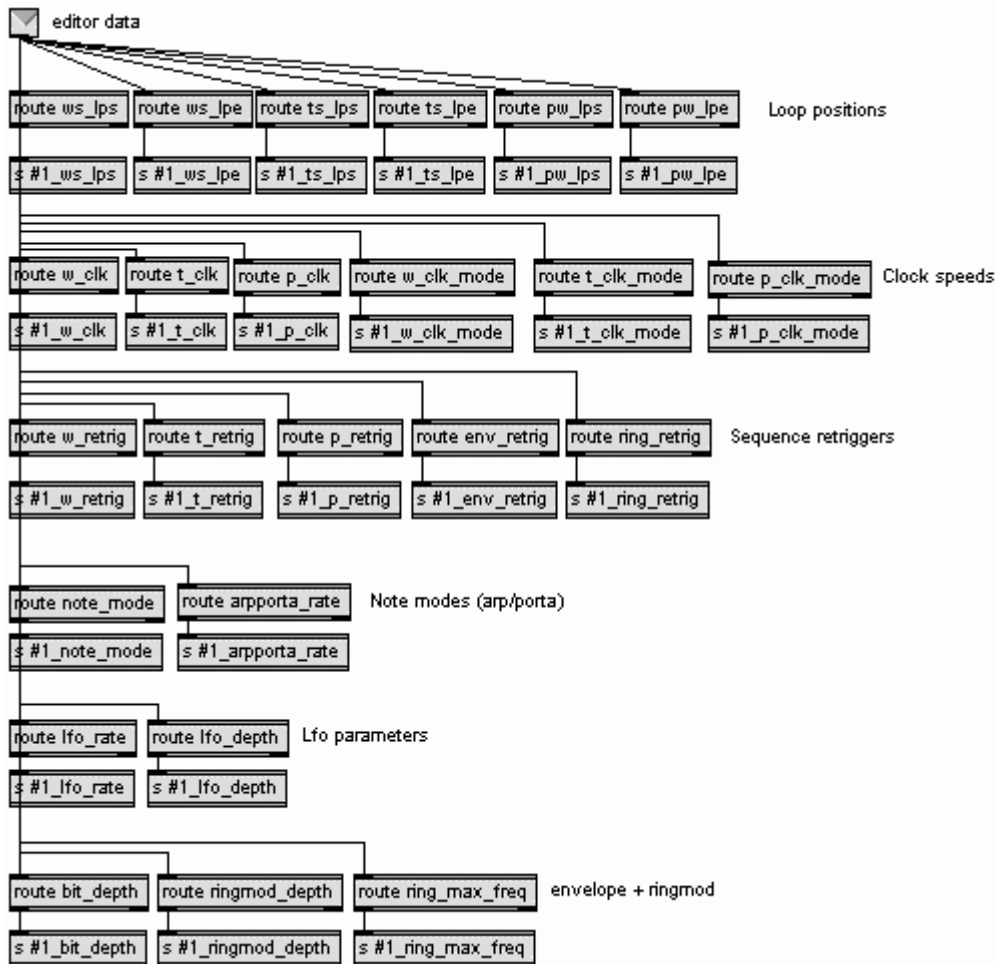


Figure 5.37 - Final [transportdata] abstraction MaxMSP structure

Options, problems, and decisions

This structure could have been implemented directly in the [editor] abstraction, however; the decision was made to separate the user interface and the conversion of user interface data. This modular approach aided the testing and debugging process, and will enable greater flexibility for future development.

By collecting these elements into a separate abstraction, a single convenient place exists to connect message boxes to individual [s] objects sending messages to synthesis components. Similarly, [print] objects can be connected to the output of the [route] objects to confirm that the correct messages are being filtered.

5.6.2 [convertdata] abstraction

Section 5.5.1 , beginning on page 96, explains that the [convertdata] abstraction is used to convert the sequence of values stored in [mslider] objects, to wavetable data stored in [buffer] objects. Figure 5.38 below shows the internal structure of the [convertdata] abstraction.

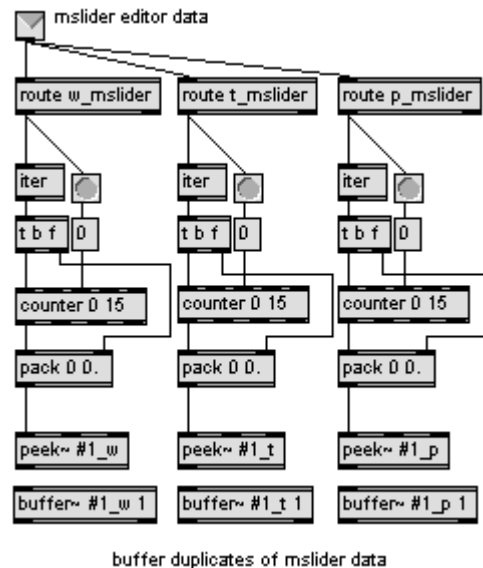


Figure 5.38 - Final [convertdata] abstraction MaxMSP structure

A brief overview of the functionality of this structure is that the [iter], [t] and [counter] objects provide a way to step through the list of values stored in the [mslider] objects in the user interface. Each value is given an index number and is written into a [buffer] object in sequence. The buffers are given unique identifiers based on the “#1” token, as explained before.

This component forms an element of an “interface map” or “bridge” explained in Figure 1.5 on page 9. As such, it is related closely to the design of the [clockedseq~] abstraction in the synthesis structure.

Options, problems, and decisions

As mentioned previously, the [convertdata] abstraction was made to satisfy the original (and regrettably unimplemented) objective of allowing step sequence data to be replayed at audio rates. The conversion of [mslider] control-rate information to signal-rate [buffer] objects is strictly unnecessary in terms of the current specification, since the [clockedseq~] abstraction does not implement audio rate playback. If more time was available, the author would implement a way to use MIDI note pitch (possibly scaled by multiplying factor) to the playback speed of a step sequence. The custom [cindex~] external (see section 5.8 on page 112) could be used for this purpose.

Since audio rate playback of the wavetable [buffer~] objects is not implemented, the [convertdata] abstraction could have used [coll] objects to store the [mslider] data. This would require the [clockedseq~] abstraction to have an internal structure more like the much like the [midiarpeggiator~] abstraction (see section 5.3.3 on page 70).

The [counter] object was chosen due to the (again, unimplemented) possibility of allowing the user to alter the length of the [mslider] step sequence as mentioned in section 5.5.2 (page 97). Implementing this feature would involve omitting the fixed limits specified in the arguments of the [counter] objects.

5.7 Improving computational efficiency

As discussed previously, the MaxMSP programming environment allows the construction of all sorts of graphically programmed structures, however; it is possible for graphically programmed structures to be inefficient in terms of computation time. A C-language compiled external object has the potential to outperform graphically programmed structures.

Also discussed previously: an interesting area of potential development is the playback of step sequences at audio rates, coupled with the dynamic routing of these signals by a central modulation matrix, itself controlled by audio-rate step sequences. This is described in section 5.3.1 beginning on page 60.

During the testing procedures, it was found that the entire system appeared to be quite inefficient in terms of the CPU usage. The development laptop supports the Intel SpeedStep technology, described by Intel as “dropping [the processor speed] to a lower frequency by changing the bus ratios and voltage, conserving battery life.” (Intel, 2003)

It was noted that when the CPU was under clocked and the project was loaded with the MaxMSP synthesis structure sitting idle (with the MaxMSP audio processing enabled); the CPU usage of the MaxMSP system hovered between 40 and 50%. This value did not change much even when all four synthesis channels were “active”. The author first thought that this was due to the complexity of the [clockedseq~] abstraction, since this is where most of the conversion between control-rate and signal-rate values occur. Also, the [clockedseq~] abstraction constantly sends control-rate signals to update the sequence position values in the user interface.

Strangely; when the audio processing in MaxMSP was disabled, the CPU usage reduced to a value between 2 - 8%. This seems to suggest that the rapid exchange of control-rate signals between modules is not the cause of the high CPU usage after all.

Returning to the idea of replaying step sequences at audio rates: several attempts were made to graphically program a structure that would provide audio rate playback (see clocked seq problems, groove, etc). Intent on exploring the possibilities of adding this audio-rate playback of step sequences, and presented with the possibility of reducing the complexity of the [clockedseq~] abstraction in the process; the author investigated the process of constructing a custom C-language external with the MaxMSP SDK (Cycling'74, 2006b) to replace the [clockedseq~] abstraction. This replacement would be called the [cindex~] abstraction, named after the [index~] object to read values from wavetable buffers with an added “c” to indicate that the object has an internal counter.

It was the intent that this be the first in a series of externals to improve the efficiency of the system. However, the time taken to design, implement, and debug the [cindex~] external proved to be much longer than expected. Eventually, development of the external was abandoned, despite the advantages afforded by the audio-rate playback possible when using

the [cindex~] external to drive the step-sequence playback. Commented sourcecode for the [cindex~] external is included in the appendices on page 138.

During development of this external, the computational efficiency was compared to the graphically programmed [clockedseq~] abstraction, and no significant improvement was detected. This result confirms the suspicion that signal-rate computations are the cause of the increased CPU usage.

Since the CPU usage only increased when MaxMSP audio was enabled, the inefficiency must be coming from areas that use signal-rate values: Also, since improving the efficiency of the step sequence playback offered no improvement, the efficiency of other elements of the synthesis structure were therefore examined.

Unfortunately, the MaxMSP programming environment lacks any tools that allow a programmer to trace or “profile” the amount of CPU time a specific area of a MaxMSP structure is using. In order to narrow down the area of the synthesis structure that might be taking up so much CPU time the author attempted to manually profile the synthesis structure, by removing sections of the synthesis structure selectively, taking note of the CPU time for each removed or modified component. The average CPU time values were generated using all four channels of the synthesis structure, averaged over a period of 10 seconds. Table 5.2 shows the results.

Configuration description	Avg. CPU %
-Default configuration -	33
[arpporta~] abstraction removed	31.5
“Pitched” noise generator removed	30.5
All [clockedseq~] abstractions removed	28.5
All [clockedseq~] replaced with [cindex~]	28.5
“Sampled” noise generator removed	28.5
LFO structure removed	28.5
Square and sawtooth oscillators removed	28
Triangle oscillator removed	27.5
Amplitude envelope and LFO removed	26.5
Ringmodulation structure removed	26.5
Amplitude envelope structure simplified	25.5
Amplitude envelope removed	21

Table 5.2 - Results of synthesis structure efficiency tests

The results indicate that the amplitude envelope is the most inefficient element, however; even completely removing this element only produced a 12% change, since four channels were active during the test this is a performance increase of less than 3% per channel.

Using the speed-step utility to adjust the CPU speed to a “normal” speed of 1.6GHz (Gigahertz), the CPU time used by MaxMSP with all four channels active averages at around 10%.

5.8 Custom MaxMSP external

This section presents a brief account of the design and implementation of a custom external object for the MaxMSP system. The external was designed, implemented, and tested, but due to problems encountered during the testing time restraints prevented the external from being integrated into the final solution.

The inclusion of the untested external into the final solution could have caused problems in other areas of the system, invalidating previous tests and requiring more tests to be carried out. Given the obvious time restrictions of the project, the author decided against the inclusion of the external. Development of the external will continue in the author's spare time after the project has been submitted.

A great deal of effort and time was spent learning how the SDK (Cycling'74, 2006b) sets out structures for programming a MaxMSP external and how certain rules must be followed. The author gained a great deal of knowledge and confidence during development. Even though the external was not integrated into the final solution, the time invested was not wasted.

Source code for the external is provided in an appendix. The source code is fully commented by the author, which should go some way towards showing the author's understanding. The placement of the source code in an appendix should not be interpreted as an indication that it is less important than the main text. The reader is strongly encouraged to read this commented code.

5.8.1 Requirements specification

An alternative to the control-rate playback of step sequence information to the synthesis structure is outlined in the main report under the section heading “[clockedseq~] abstraction”. The main report also describes the attempt (and failure) to build a structure from standard primitive MaxMSP objects that steps through wavetable buffer, incrementing the read index position at regular intervals.

The main report also described the requirement that the read index positions should loop between specific index points, and that the structure should support extremely small intervals such that the rapid output of the wavetable values form audio frequency oscillations.

The requirements of this component, summarised, are therefore:

- Access a wavetable buffer of values
- Advance through the wavetable values at a rate specified in milliseconds per wavetable-sample
 - Step through the wavetable positions one at a time, holding the output at the value stored in the wavetable for a specified number of milliseconds before advancing to the next wavetable position.
 - Support extremely small millisecond intervals, such that the output becomes a stream of wavetable values oscillating at audio frequencies
 - Optionally specify the oscillation frequency in Hz?
- Allow looping between buffer index positions
 - Cycle between “loop start” and “loop end” wavetable index read positions

- Jump to the start of the wavetable if a “note trigger” message is received
- Provide signal-rate output of the wavetable values
 - Optionally provide a control-rate output to update the user interface with an indication of the current index position in the wavetable

5.8.2 Implementation

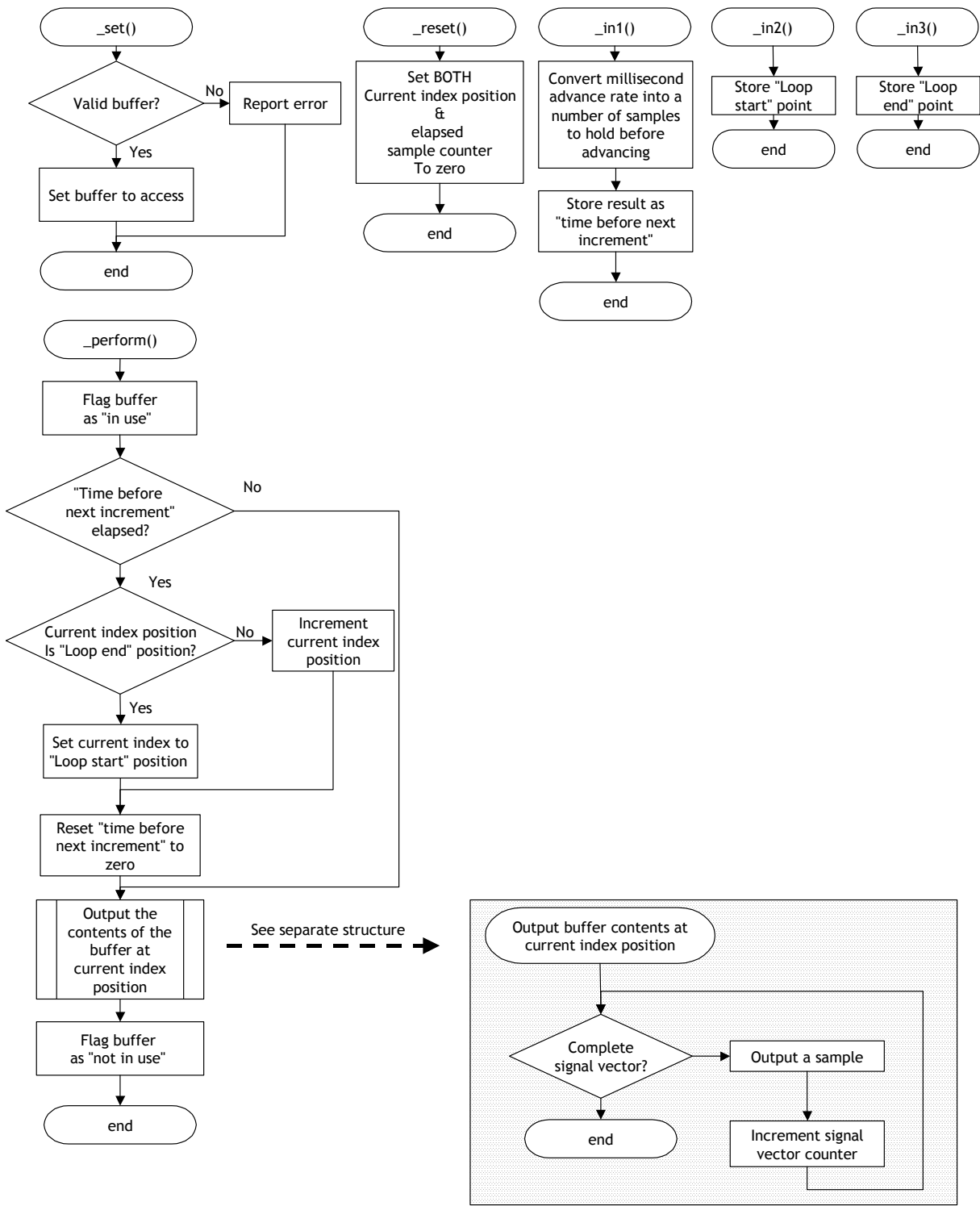
The MaxMSP SDK includes several example projects designed to be used with the Microsoft Visual C++ IDE. One of these examples is the source code for the standard [index~] object which reads values from a wavetable buffer. The source code for the [index~] example project was altered to conform to the specification.

The original [index~] example sourcecode describes a simple structure that uses the value described by a signal-rate input into a “read index pointer” to read a specific value from the wavetable buffer. By modifying the [index~] example sourcecode slightly to include an internal counter to provide the read index positions instead the [index~] external was modified to form the [clockedindex~] object.

Two counters were created, one that is used as a read index position for accessing the wavetable buffer. A second counter monitors the number of elapsed samples since the last increment of the read index position counter.

To provide the looping between index positions, the read index position counter’s upper and lower limits are checked after each adjustment to the buffer read index position.

A flowchart of the object functionality is included below. Not all functions described by the source code are shown in the flow chart (for example, the new object and dsp initialisation routines required by MaxMSP), however all functions outlined in the requirements are included as flow chart representations.



Comparing to the requirements specification:

- _set() Determines the wavetable buffer to access
- _reset() Jumps to the start of the wavetable if a “note trigger” message is received
- _in1() Determines the interval between wavetable index position increments
- _in2() Determines the “loop start” index position
- _in3() Determines the “loop end” index position
- _perform() Steps through the wavetable at the defined interval, between start and end loop points

5.8.3 Problems and abandonment

The structure in the flow chart, which is in charge of the output, uses the term “signal vector”. This section explains the term, and the problems encountered during development that eventually led to the abandonment of the custom external.

MSP objects must produce a signal-rate output for at a fixed sample rate (the default being 44,100 samples per second) determined by the MaxMSP host application. The MaxMSP host software deals with signal-rate samples in chunks called “signal vectors”. The signal vectors provide a buffered approach to signal generation and processing, theoretically eliminating glitches if the operating system on which the MaxMSP software is running takes priority away from audio processing for some reason.

Due to the author’s inexperience with MaxMSP external programming, combined with the highly variable quality of documentation provided as part of the SDK. The author was unable to overcome certain problems related to the signal vector size and the minimum interval between advancing to a new index position in the wavetable buffer.

These problems prevented the user from specifying a step interval smaller than the signal-vector size. A user who selects large signal vector sizes (e.g. 2048 samples, admittedly very high for most “normal” systems) will be prevented from specifying a wavetable buffer advance interval of less than 46 milliseconds. The author could see no way around this problem due to lack of detail in the SDK documentation.

Given that the time remaining until the deadline was short, and that many tasks in other areas of the project were still unfinished; development of the external was abandoned.

Chapter 6 Conclusions

This chapter summarises the work done, the report contents, potential future developments, the project objectives (completed and uncompleted), learning achievements, and evaluate the chosen time/project management strategies.

Putting together all the information presented, it should now be clear how the entire system operates; from the input of data at the user interface, to the communication of this data to the synthesis structure, to the operation of the synthesis structure itself.

In conclusion, a software synthesiser has been designed, produced, tested, and fully documented to the best of the author’s abilities in the time allowed. The project has enabled the author to develop a wide range of skills; in certain cases new skills (such as the user interface design) were acquired and developed from scratch over the course of the project.

Initially, the user interface was too fragmented, requiring multiple windows to be opened revealing the parameters of a single channel. When multiple channels were to be configured, the user could easily be confused by the similarity of the various windows. The decision was made to combine these into a single interface per channel was made in order to simplify the user interface.

It is important to restate the benefits of including MIDI zone processing. These were initially included to enable the synthesiser to be used more easily in conjunction with a MIDI

sequencer, the MIDI zones replace the functionality of instrument numbers next to note triggers that are used in a tracker but are missing in most MIDI sequencers.

The benefits of MIDI zone processing also extend to live performance. It would be possible to configure several (possibly overlapping) zones that enable a performer to play simultaneous percussion parts, arpeggiated chords, and melodies simultaneously. A controller keyboard with a suitable range of octaves would have to be used and the performer would need to be practiced in stretching between octaves, but it could be done.

The author developed useful techniques for managing large projects, along with an understanding of the LaTeX/LyX documentation system and how to apply the chapter/section/subsection outlining tools provided as part of the Microsoft Word wordprocessor in a similar way.

Valuable and transferable skills were developed during the development of a custom external MaxMSP object written in the C programming language. More information about this can be found in section 5.8. beginning on page 112.

The research carried out into the internal structure of sound chip hardware has been a wonderful opportunity to expand a simple curiosity that the author had into a clear understanding of how digital hardware synthesis and software control structures can work in collaboration. Without this project as a catalyst, the author probably would not have developed the crucial independent learning skills required to research a topic in such depth.

It should be stressed that the appendices include valuable information about the internal structure of several hardware sound chips. This information was compiled with great care by the author from many datasheets and circuit diagrams. Its inclusion in an appendix should not be interpreted as an indication that it is less important than the main text; it is simply placed there to avoid breaking the flow of the text. The reader is strongly encouraged to read this information, and to evaluate it as part of the report.

6.1 Content summary

A brief summary of the report content follows. This report has:

- Introduced the problem of composing chiptunes with modern synthesisers and sequencers
- Characterised the desirable qualities of chiptune timbres
- Introduced the concept of a graphical programming environment such as PureData and MaxMSP
 - Explained various MaxMSP concepts such as objects, patches, and abstractions were also covered
- Defined the objectives and requirements that the project must meet
- Emphasised the great detail in which the decisions behind the strategies devised to meet the objectives, in particular:
 - project management strategy
 - report structure strategy
 - system design strategy (of individual modules and the integrated whole)
 - software implementation and development strategy
 - software testing strategy
- Introduced the concept of MIDI to enable readers who are unfamiliar with it to be more comfortable with related topics discussed in the report.
- Classified the differences between MIDI sequencers and Trackers,
 - This was achieved by viewing the historic computer systems and sound chips as "layered systems", drawing parallels with the OSI 7-layer model.
- Determined the most suitable graphical programming environment for designing a user interface, and:
 - Stated the reasons for choosing MaxMSP as the graphical environment based on the examination of the available options
- Presented the research carried out into assessing the suitability of various user interface designs popular in historic systems for chiptune composition
 - Defined the influence certain interface designs have on the user
 - Evaluated the benefits and drawbacks of certain interface designs in historic chiptune composition software
 - Acknowledged several desirable interface techniques and features from modern software sequencers and synthesisers
 - Acknowledged several desirable properties of interfaces from historic chiptune composition software
- Explained the benefits of tracker interfaces in terms of a single unified solution to chiptune composition
 - Identified and presented possible solutions to certain problems associated with separating the interface for sound design (implemented as part of the project) from the interface for the composition of notes (as an external sequencer software entity).
 - Considered other problems relating to the storage and organisation of sound parameter data.
- Investigated the possibility of a genetic algorithm for the semi-autonomous creation of sound parameter data.
 - Examined modern interface designs for allowing the user to control this process.

- Presented the detailed research carried out into the synthesis methods involved in chiptunes.
 - Separated the synthesis into hardware and software elements, this separation is influenced by the analogy of "layered" systems introduced earlier.
 - Emphasised the importance of this separation for implementing an efficient system.
 - Included information about the structure of historic hardware sound chips in an appendix. This information was extremely valuable during the design of the synthesis structure.
 - Indicated the existence of software with similar goals to the project
 - Acknowledged several useful features present in these systems worth including as part of the project
- Summarised the findings of the research
- Summarised any changes and additions to the project objectives and requirements as a result of information obtained during the research
- Explained the concept of the entire system and how each module fits together
 - This was approached in a top down manner, examining the entire structure in gradually increasing levels of detail.
 - Included an explanation of commonly used MaxMSP objects and concepts to help readers who are unfamiliar with the environment understand better.
 - Divided the explanation into three sections which reflect the three main components of the system design: Synthesis, Interface, and Parameter storage/communication
- Provided detailed notes of the implementation for each module:
 - Outlined the requirements and purpose of each module
 - Detailed any options, problems, and decisions made
 - Emphasised any changes to the initial design as a result of prototyping
 - Explained the final solution by referring to the final MaxMSP structure
 - Indicated any potential future developments for each module
 - Included test data, results, and recommended bounds for inputs and outputs for each module to aid software maintenance tasks
- Assessed the computational efficiency of the system
 - Explained the procedures used to achieve these results
 - Considered possible improvements to the system
- Detailed the development of a custom MaxMSP "external" object

6.2 Time planning

Particular attention should be drawn to the development of a custom external MaxMSP object, written in the C language, to replace the problematic graphically programmed wavetable playback that forms part of the synthesis structure. Section 5.8 on page 112 outlines the development of this external object.

The author feels that much time was "wasted" developing graphically programmed structures to access wavetables, when this functionality was simply and quickly implemented in the external. If the external development had begun earlier, more time could have been devoted to developing the intended synthesis structure discussed in and section 5.3.1 beginning on page 60 involving a modulation matrix for audio-rate control parameters.

The decision of the supervisor to hold 1 hour project meetings every two weeks, as opposed to 30 minute meetings every week meant that an issue requiring the attention of the supervisor arising after a project meeting could mean that two weeks would be wasted before the issue could be mentioned in the next project meeting with the potential for four weeks to elapse this way if the issue requires an additional meeting to resolve.

In the very early stages of the project, deciding a project topic took a long time. The author feels that the decision to choose this topic of chiptune synthesis was not a bad one however.

The preparation of the final report was especially well executed. Section 1.5.4 beginning on page 15 outlines the decision to abandon a formal documentation procedure during the entire project, instead favouring a project log book. By choosing to start this log book at the very early stages of the project, the preparation of the final report was made much easier; enabling many revisions of the final document were able to be made before submission.

With regard to time planning in general; the Gantt chart features several scheduled review points which were not carried out. The intention was for these review points to involve a formal evaluation of progress and a new revision of the Gantt chart to be made based on the remaining tasks.

Instead, however; the adopted technique was to keep and update a to-do list as part of a project log book. These lists determined the time spent on each task, whilst referring to the initial Gantt chart (see appendices) to gain an overall impression of the number of weeks remaining.

The initial Gantt chart features several weeks of "slack time" before the submission date. The author is pleased to report that this slack time was largely unused. The project implementation was complete on schedule, with the documentation taking slightly longer by only one week.

6.3 Further work

Although certain objectives were not met (e.g. the modulation matrix with audio-rate parameter sequences, genetic algorithm for parameter breeding, and the drag and drop user interface) the flexibility of the system has enabled many changes to take place without a complete re-design.

The fact that the system design has gone through many revisions is an indication of the strong flexibility of the original design concept. Despite these modifications to the original design, several improvements could still be added:

- Audio-rate playback of parameter sequences via continued development of the custom external
- Modulation matrix to control parameter sequence destination
 - Modulation matrix controlled by parameter sequence with feedback path
- Re-structuring of the interface to configure parameter sequences
 - The current interface divides parameter sequence entry into separate areas
 - Visualisation of the interplay between separate parameter changes is difficult

- Display of the parameter sequences as a composite overlay might help enable more complex sounds to be designed where the interplay between different parameter changes is key.
- Abandonment of envelope/breakpoint function parameters may help with the design of sounds where synchronised sequences are required. (The current interface makes it difficult to generate sudden amplitude steps, requiring many mouse clicks)
- Dynamic re-sizing of parameter lengths.
 - A user should not be limited to 16 steps per parameter sequence, and the [mslider] object supports the appropriate graphical scaling for longer sequences.
- Re-ordering of parameter sequence values
 - By cyclic rotation "shuffling" of values
 - By loading and saving individual parameter sequences to disk
 - By dragging and dropping parameter sequences between areas (EG: transposition sequence dragged to pulse width sequence)
- Midi note pitch to scale the playback speed of a step sequence for wavetable playback
- Optional arpeggiation speed adjustment by scaling playback speed of arpeggiation [coll] table
 - Using this option would ensure that the apparent duration of an arpeggiation sequence lasts a fixed number of milliseconds, no matter how many notes need to be played during this time.
- The computational efficiency of the entire system could be evaluated in more depth with proper profiling tools
 - Optimisation could be carried out in the most needed areas.
- Lifting of the 4 channel limitation.
 - One option would be to specify any number of channels (up to a sensible, but very high limit).
 - A practical channel limit would no doubt be enforced by the computational processing power of the host machine.
- Audio output recorder to enable integration with multitrack sequencers.
 - This would enable users to layer more than 4 channels, optionally adding effects (delays, reverb, group compression of bass and percussion...etc)
- Optional "TV speaker" simulation for users wishing to obtain a high degree of authenticity in the output since chiptunes were often played on home computer systems connected to standard television sets via RF cables.
 - An investigation could be conducted into the frequency response of an average 1980s TV speaker, and the nature of the subtle audio signal distortion in typical RF modulation and demodulation circuits.
- A further investigation could be carried out into pseudorandom noise

- More specifically, by investigating the methods employed in the historic sound chips such as shift registers and XOR gates in a feedback configuration.

Some of the suggested improvements above were investigated during the course of the project, but these were not implemented either due to lack of time to test the solution with enough confidence to include in the final version, or due to lack of time to develop the solution beyond a simple prototype.

6.4 Fulfilment of objectives

6.4.1 List of completed objectives

6.4.2 List of uncompleted objectives

x and y were investigated but not implemented. author did not feel confident enough in the experimental design to allow them to be part of the final system: too many bugs/glitches/things to add for complete compatibility of “older” way.

6.5 List of achievements

6.5.1 Project achievements

1. Single sentences
2. Numbered list

6.5.2 Learning achievements

1. Single sentences
2. Numbered list

References

- Arbiter.co.uk, 2007 *Native Instruments FM7 screenshots*, [online] available from http://www.arbiter.co.uk/ni/products/images/fm7_library.gif. accessed 01/03/07
- BULBA, S.V., 2000 *Vortex Tracker II v1.0 beta 13 (software)*, [online] available from <http://bulba.at.kz/>, accessed 26/12/2004
- CAREHAG. J. & FREDRICKSON. J., 1995 *MusicLine MLine v1.15 (software)*, [online] available from <http://www.musicline.org/>, accessed 09/01/2007
- Clavia.se, 2006 *Nord modular screenshots*, [online] available from <http://www.clavia.se>, accessed 15/11/06
- Comp.sys.sinclair newsgroup, 2000a *Agent X/Chronos Music* [online] available from <http://groups.google.co.uk/group/comp.sys.sinclair/msg/fa90e1112a14a233>, accessed 07/01/07
- Comps.sys.sinclair newsgroup, 2000b *Best effect in a Spectrum game?* [online] available from <http://groups.google.co.uk/group/comp.sys.sinclair/msg/daf35ca77e629647>, accessed 15/12/05
- Creamhq.de, 2007 *JAM (just another musicplayer) v2.0 beta (software)*, [online] available from <http://www.creamhq.de/jam.php>, accessed 01/04/2007
- CREASEY. D., 2005 *Audio processing course notes*, Bristol: University of the West of England.
- Crossfire-designs.de, 2007 *A retrospective view on sound card history*. [online] available from <http://crossfire-designs.de/index.php?lang=en&what=articles&name=showarticle.htm&article=soundcards&print=true>, accessed 17/01/2007.
- Cycling'74, 2005 *Max4.5 Reference manual*, Cycling'74/IRCAM
- Cycling'74, 2006a *Cycling'74 homepage* [online] available from <http://www.cycling74.com>, accessed 01/11/06
- Cycling'74, 2006b *MaxMSP SDK*, [online] available from <http://cycling74.com/download/maxmspSDK.zip> (accessed 01/04/2007).
- Delitracker.com, 2007 *Delitracker (software)*, [online] available from <http://www.delitracker.com/>, accessed 01/02/07

FELDMAN, M., 2007a *Programming the PC Speaker. PC-GPE collection* [online] available from <http://www.qzx.com/pc-gpe/speaker.txt> (accessed 17/01/2007)

FELDMAN, M., 2007b *Programming the Intel 8253 Programmable Interval Timer. PC-GPE collection* [online] available from <http://www.qzx.com/pc-gpe/pit.txt> (accessed 17/01/2007)

FOLLIN, T., 2007 *Dr. Follin's home surgery* [online] available from <http://www.timfollin.pwp.blueyonder.co.uk/Biog2.htm>, accessed 21/01/07

Hard Software, 2003 *HardSID - The SID 6581/8580 (C64) MIDI Synthesizer SoundCard for PC*, [online] available from <http://www.hardsid.com>, accessed 30/03/2007

Homeftp.net, 2007 *GoatTracker v2.59 (software)*, [online] available from <http://cadaver.homeftp.net/>, accessed 14/03/2007

HUBER, D.M, 1999 *The MIDI manual* (2nd ed), Boston Mass.: Focal Press - Butterworth Heinemann.

Individual computers, 2007 *Catweasel* [online] available from http://jschoenfeld.de/products/products_e.htm, accessed 30/03/2007

Intel, 2003 *Intel speedstep technology* [online] available from <http://www.intel.com/support/processors/mobile/pentiumiii/sb/CS-007509.htm>, Accessed 05/04/2007

LAUSTSEN. B. N., 2001 *CyberTracker v1.01 (software)*, [online] available from <http://noname.c64.org/tracker/>, accessed 21/11/2006 (CYBERBRAIN/Noname)

LAUTENBACH, F., 2006 *MusicMon v2.5d (software)*, [online] available from http://dhs.nu/files_msx.php, accessed 10/01/2007 (DARK ANGEL/Aura)

Littlesounddj.com, 2007 *LittleSoundDJ v3.7.4 (software)*, [online]. available from <http://www.littlesounddj.com/lsd/>, accessed 01/04/2007

LYON. E, 2006 *Lyon's Max/MSP potpourri of externals (software)*, [online] available from <http://arcana.dartmouth.edu/~eric/MAX/>, accessed 21/02/2007.

Lyx.org, 2007 *LyX (software)*, [online] available from <http://www.lyx.org/>, accessed 01/02/2007

Matthes, O., 2005 *Flashserver (software)* [online] available from <http://www.nullmedium.de/dev/flashserver/>, accessed 13/10/06

Mda-vst.com, 2007 *MDA JX10 (software)* [online] available from <http://www.mda-vst.com>, accessed 07/01/2007

Midi.org, 2007 *MIDI manufacturers association* [online] available from <http://www.midi.org/about-midi/gm/gm1sound.shtml>, accessed 22/02/07

Mmonoplayer.com, 2007 *mmonoplayer buza*, [online] available from <http://www.mmonoplayer.com/mspexternals.php>, accessed 17/01/2007

MORRIS, G., 2006 *MaxYMiser v1.19 (software)*, [online] available from <http://www.preromanbritain.com/maxymiser/>, accessed 21/01/2007

MORRIS, G., 2007 Personal communication.

- NASH, C., 2004 *VSTrack, Tracking software for VST hosts*. [online], M.Phil thesis, Dublin trinity college. Available from <http://www.nashnet.co.uk/english/reViSiT/documentation.htm>, accessed 21/11/2006
- PENFOLD, R.A, 1995 *Advanced MIDI user's guide* (2nd ed),Tonbridge: PC Publishing.
- Puredata.org, 2006 *About Puredata - PD community site* [online] available from <http://www.puredata.org>, accessed 01/11/06
- Refx.net, 2007 *reFX - VST/AU Instruments* [online] available from <http://www.refx.net>, accessed 07/01/07
- Rf1.net, 2006 *Miditracker v1.2* (software), [online] available from <http://www.rf1.net/software/mt/>, accessed 01/03/2007.
- ROADS. C., 1996 *The computer music tutorial*, London: MIT Press.
- SARLO., J., 2006 *GriPD: Graphical interface for Pure Data* [online] available from <http://crca.ucsd.edu/~jsarlo/gripd/#platforms>, accessed 07/10/06
- SEEMANN, F., 1997 *Magic Synth v2.5* (software), [online] available from http://dhs.nu/files_msx.php, accessed 10/01/2007. (TAO/Cream)
- Shoodot.net, 2007 *Famitracker v0.0* (software) [online] available from <http://famitracker.shoodot.net/>, accessed 17/01/2007
- SOMMERVILLE. I., 2001 *Software Engineering* (6th Edition), Harlow: Addison-Wesley
- Steinberg, 2001 *Cubase VST32 v5.0* (software)
- TOLHURST, M. ed., 1988 *Open systems interconnection edited by Mark Tolhurst*, Unknown: Macmillan.
- TOYOSHIMA, T., 2004 *T'SoundSystem* [online] available from <http://www.toyoshima-house.net/tss/>, accessed 30/12/2005
- Tuwien.ac.at, 2007 *Interview with Martin Galway from Commodore Zone (typed by Adam Lorentzon)* [online] available from <http://stud1.tuwien.ac.at/~e9426444/mgalway.html>, accessed 09/01/2007.
- VIDOVIC, A., 1992 *YMTracker v1.0* (software), [online] available from http://dhs.nu/files_msx.php, accessed 10/01/2007
- WITTCHOW, O., 2007 *Nanoloop v1.3* (software), [online] available from <http://www.nanoloop.com>, accessed 01/04/2007
- WODOK, M., 1998 *Abys's Higher Experience AHX v2.3* (software), [online] available from <http://www.amigau.com/amigarealm/ahx/main.html>, accessed 06/03/2007 (DEXTER/Abys)

Bibliography

- Benchun.net, 2006 FLOSC website, available online <http://www.benchun.net/flosc/>, accessed 14/10/06
- RUSS. M., 1996, *Sound synthesis and sampling*, Oxford: Focal press

Appendix 1
Completed Proposal form

Appendix 2
Completed progress report form

Appendix 3
Completed progress monitoring form

Appendix 4

Sound chip hardware research

<p>TIA Television Interface Adapter (Atari 2600)</p>	<ul style="list-style-type: none"> • 2 Tone oscillators (square wave) • 2 Noise-Tone oscillators (periodic pseudorandom number generator) • Registers for controlling pseudorandom periodicity, square frequency, and amplitude • The amplitude register is 4bits (taking values 0-15) • The frequency register is 5bits (taking values 0-31), each oscillator operates by dividing a 30kHz reference clock. The output of this divider network is used to clock the noise and tone generators. • The periodicity register is 4bits (taking values 0-15)
<p>Unusual aspects</p>	<ul style="list-style-type: none"> • The periodicity of the pseudorandom noise can be altered. • Pseudorandom noise is generated by a 9bit shift register arrangement with feedback taps (see Figure A4.1 below). Different values in the periodicity register select the feedback taps to change the character of the noise. • Some compositions combine certain square wave frequencies with periodic pseudorandom noise where it appears that two tones form a third harmonic “beating frequency” or “wolf tone”. <p>Figure A4.1 - pitched tone noise generator logic (Atari, 1982)</p>

Compiled from circuit diagrams (Atari, 1982a), information by Mark De-Smet (2007), and from documentation available on Mmonoplayer.com (2007).

<p>SN76489 (various revisions including SN76496)</p> <p>(Acorn BBC, Tomy Tutor)</p> <p>Texas Instruments</p>	<ul style="list-style-type: none"> • 3 square wave oscillators • 1 noise generator (periodic or continuous pseudorandom number generator) • Individual 4-bit attenuators for each sound generator.
--	---

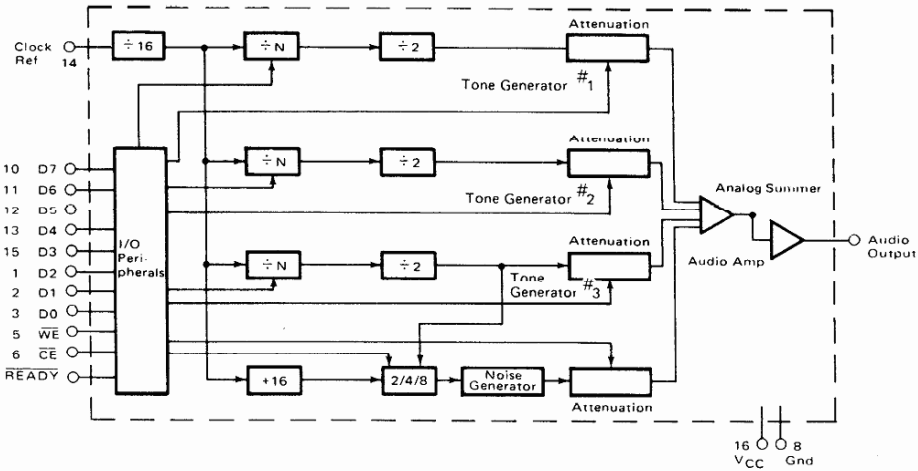
<p>Unusual Aspects</p>	<ul style="list-style-type: none"> • Sega duplicated the basic functionality of the SN76489 into their Sega Mastersystem, Gamegear, and Megadrive (AKA Genesis in the USA) games consoles. These Sega versions are not exact clones and, among other differences, exhibit subtle differences in the pseudorandom number generator. • The individual attenuators for each of the oscillators (3 square and 1 noise) allow for unique and complex envelope shapes to be generated externally by the CPU. • Nonlinear DAC 
------------------------	---

Figure A4.2 - block diagram from datasheet (Texas Instruments)

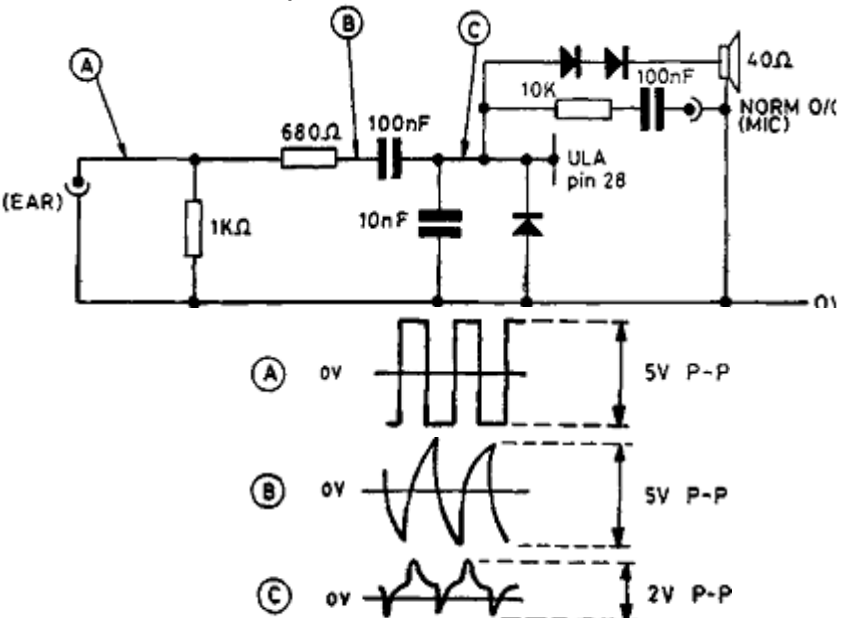
Compiled from the SN76489 datasheet (Texas instruments, unknown year), Howell1964 (2006), Kortnik (2006), Floodgap.com (2007), Smspover.org (2006) and Wikipedia (2007a).

<p>POKEY</p> <p>(Various arcade machines)</p> <p>Atari</p>	<ul style="list-style-type: none"> • 4 square wave oscillators • 3 noise generators with variable periodicity • Volume registers control output current • Various clocking rates • Various 8/16/12 bit output combinations
<p>Unusual aspects</p>	<ul style="list-style-type: none"> • Noise generators are fed into a “Sample and Hold” circuit that can be clocked at different frequencies. This “sampled noise” produces a very characteristic sound unique to the POKEY chip.

Compiled from the POKEY datasheet (Atari, 1982b), also Leopardcats.com (2007).

<p>K051649 (SCC)</p> <p>(used in some MSX cartridge games)</p> <p>Konami</p>	<ul style="list-style-type: none"> • 5 channels of 8-bit wavetable replay • Wavetable length for each channel is 32-bytes • Individual 4-bit volume registers, and 12-bit playback frequency registers for each channel
<p>Unusual aspects</p>	<ul style="list-style-type: none"> • Channels 4 and 5 must share the same waveform, but may have different volumes and playback frequencies. • Individual volume registers for each channel allow for complex CPU generated envelopes. • A “deformation” mode causes the wavetable data to rotate in the buffer. • Wavetables may be updated dynamically by the CPU.

Compiled from Hansotten.com (2007) and Wikipedia.org (2007b).

<p>ZX Spectrum (48k model)</p> <p>Sinclair Research Ltd.</p>	<ul style="list-style-type: none"> • Very unsophisticated sound hardware • Pin 28 on the spectrum custom chip is a dual purpose 1-bit DAC/ADC pin  <p>Figure A4.3 - ZX Spectrum 48K version (Sinclair Research Ltd, 1984)</p>
--	--

Compiled from schematics Sinclair Research Ltd (1984).

SID (various revisions including 6581R2, 6581R3, 6581R4AR, 6582A, 8580R5)

(Commodore64)

Commodore

- 3 oscillators (range 0Hz-4kHz)
- Each oscillator waveform switchable between Variable pulse, Sawtooth, Triangle, Noise (i.e. pseudorandom number generator)
- Programmable analogue filter (12dB/8ve, LP,HP,Notch variable between ~30Hz-12kHz, variable resonance)
- 3 (one per oscillator) exponential ADSR envelopes, with attack response from 2ms-8s, decay and release time from 6ms-24s.
- Programmable ring modulation (output of an oscillator can modulate the amplitude of the remaining two by 48dB)
- Oscillator synchronisation (output of an oscillator can reset the internal phase angle of another)

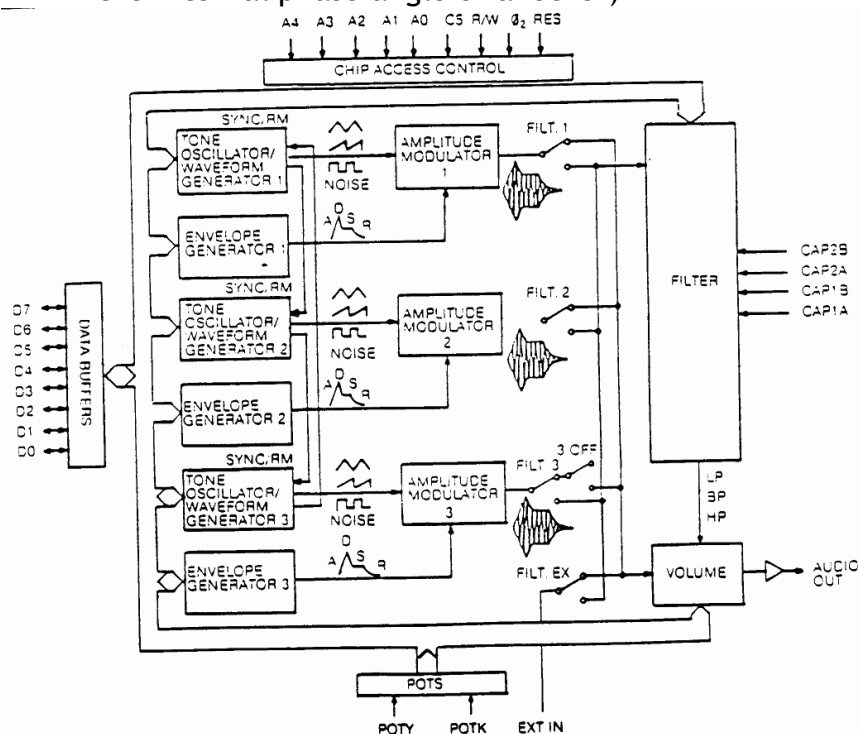


Figure A4.4 - 6581 block diagram (Commodore, 1982)

Unusual aspects

- High NMOS signal leakage in the 6581 chips meant that adjusting the envelope sustain level can produce “pops” as internal DC offsets are scaled to produce sudden jumps in output. This was sometimes used to produce percussion in some compositions. This is known as “Galway noise” (after Martin Galway discovered it)
- One highly unusual technique is to play back short 4-bit PCM waveforms with the aid of the CPU and interrupts causing “pops” to occur at audio frequencies.
- Abuse of “timing problems” that cause phase shifts in the Triangle and Sawtooth waveforms, combined with AND gates to mix the waves together can produce new wave shapes.
- The shift register feedback arrangement, used to produce the random numbers, takes one of its inputs from the register used to set the tone oscillator frequency. As a

	<p>result, the noise output has a different pattern for each frequency setting. This results in a very unique sounding noise channel.</p> <ul style="list-style-type: none"> • Amplitude modulators between oscillators are rarely found in other sound chips of this era • Voltage controlled resonant lowpass/highpass/notch filters are also a rare feature of the SID
--	---

Compiled from the 6581 Datasheet (Commodore, 1982), Jo'ogn (2007), Karlsen (2003), Kubarth (2006), and Tuwien.ac.at (2007).

<p>PAULA (Amiga computers) Commodore</p>	<ul style="list-style-type: none"> • No on-chip oscillators • 4 channels of hardware driven (DMA/CPU) DAC with 8-bit resolution • The four channels are mixed to an output stereo pair. • Each of the four DMA channels has a 6-bit volume register and a playback frequency register • One channel in a channel pair may modulate the other channel's period or amplitude. This is a hardware feature, not a CPU/Interrupt driven modulation of the amplitude or period registers
--	---

<p>Unusual aspects</p>	<ul style="list-style-type: none"> • The “official” maximum sampling rate for instrument playback is approximately 28,867 Hz, but since clock signals from the video hardware are used, setting a specific video blanking rate on AGA chipset machines will double the maximum sample rate to 57,734 Hz. • A sample depth of 14-bit can be created using two channels: one playing the most significant 8 bits at maximum volume while the other plays the least significant 6 bits at minimum volume. • CPU driven audio output is also possible. Rather than using the chip as a DMA device, an audio buffer is constructed in main memory and the device is used as a pure DAC, rather than a DMA sample playback device. This allows more than 4 discrete channels to be played simultaneously, but at the expense of CPU cycles.
------------------------	--

Compiled from Wikipedia (2007c), and Bel.fi (2007).

<p>2A03 (NES, SNES and extremely similar custom hardware exists in the Gameboy) Nintendo</p>	<ul style="list-style-type: none"> • 4 fixed waveform oscillators • 2 Pulse oscillators (range 54.6Hz - 12.4kHz) • 1 Triangle oscillator (range 27.3Hz - 55.9kHz) • 1 Noise oscillator (range 29.3 - 447kHz) • Fixed pulse widths (either 14%, 25%, 50%, 75% or 25% 50% 75% 87.5%) • The frequency of individual pulse oscillators can be swept up and down using a “pitch sweep” register
--	--

<p>Unusual aspects</p>	<ul style="list-style-type: none"> • SR envelopes for pulse and noise channels, triangle channel can only be “shut off” after a certain number of cycles, no envelope decay or release. (??) • The noise generator outputs a stream of pseudo random 1-bit values that can be switched between a pattern with 32,767 numbers, and one with 93 numbers. • The DAC used on the Nintendo Entertainment System is 4-bit non-linear. This non-linearity arises from the fact that, in order to output values below zero volts, a collection of 100ohm pull-down resistors are used. This produces a certain subtle distortion unique to NES music. • The NES features a separate Delta Modulation Channel (referred to in some documentation as the DMC) that is sometimes used to playback delta modulation “samples”. Some compositions use the DMC to create alternative waveforms using this method. • The DMC can also be used to playback PCM samples (as opposed to delta modulation samples) by directly writing values into the DMC output registers. • It is possible to indirectly attenuate the volume of the triangle and noise channels of the 2A03 by writing certain values into the DMC registers.
------------------------	--

Compiled from Taylor (2003a, 2003b), Slack.net (2007a 2007b), and Wikibooks.org (2007)

<p>AY-3-8910</p> <p>Also various revisions including AY-3-8912, AY-3-8913,</p> <p>Yamaha clone (YM2149)</p> <p>(MSX, various hand-held games)</p> <p>General Instruments</p>	<ul style="list-style-type: none"> • 3 square wave oscillators • 1 noise oscillator • 3 looping envelope generators • Analogue outputs via built in 5-bit DAC
--	---

<p>Unusual aspects</p>	<ul style="list-style-type: none"> • Yamaha cloned the AY-3-8910 as the YM2149. This is extremely similar to the AY series down to almost every last detail. The YM has optional double resolution (but half rate) envelope generators. • Envelope generators can be set to loop, repeating their pattern of rising or falling values. • Logarithmic DAC (see figure below) • Some compositions achieve alternative waveforms by setting an oscillator to idle at a constant DC value. Using the looping envelope generator associated with that oscillator, the static value is warped into a new waveform. <div data-bbox="443 645 1369 1254" style="text-align: center;"> </div> <p style="text-align: center;">Figure A4.5 - linear internal waveform steps result in a logarithmic analogue output (General Instruments, unknown year)</p>
------------------------	---

Compiled from AY-3-8910 Datasheet (General Instruments, unknown year), AY-3-8913 Datasheet (General Instruments, unknown year), Wikipedia.org (2007d), Yamaha (1992), and Howell1964 (2007).

Reference list (for this appendix)

Atari, 1982a, *Atari TIA schematic diagrams* [online] available from <http://www.classic-games.com/atari2600/stella.html> (accessed 4/01/2007),

Atari, 1982b, *POKEY datasheet* [online] available from <http://www.leopardcats.com/bbpokey/pokey.pdf> (accessed 28/01/2007)

Bel.fi, 2007, *For the love of Paula*, [online] available from <http://www.bel.fi/~alankila/blog/2006/01/18/For%20the%20love%20of%20Paula.html> (accessed 21/01/07).

Commodore, 1982, *6581 datasheet* [online], available from http://sid.kubarth.com/files/ds_6581.pdf (accessed 06/12/2006),

DE-SMET. M., 2007, *Mark De Smet on the Atariage website* [online] available from http://www.atariage.com/2600/archives/schematics_tia/index.html (accessed 10/01/2007)

Floodgap.com, 2007, *Tomy tutor Hardware* [online] available from <http://www.floodgap.com/retrobits/tomy/hardware.html> accessed 18/01/2007.

General Instruments, Unknown year, [online] available from <http://www.msxarchive.nl/pub/msx/mirrors/hanso/datasheets/chipsay38910.pdf> (accessed 21/01/2007).

General Instruments, Unknown year, *AY38913 datasheet* http://www.ionpool.net/arcade/gottlieb/technical/datasheets/AY_3_8913_datasheet.pdf (accessed 05/01/2007).

Hansotten.com, 2007, *MSX Hardware*, [online] available from <http://www.hansotten.com/msxhw.html> (accessed 21/01/2007)

Howell1964, 2006, *Howell1964's website* [online] <http://www.howell1964.freemove.co.uk/parts/76489.htm>, accessed 04/12/2006.

HOWELL1964, 2007, *Howell1964 website*, [online] available from http://www.howell1964.freemove.co.uk/parts/ay3891x_datasheet.htm (accessed 05/01/2007).

Jo'ogn, 2007, *Jo'ogn website* [online] available from <http://www.joogn.de/sid.html> (accessed 04/01/2007)

KARLSEN. O., 2003, *Olve Karlsen music-DSP mailinglist* [online] available from <http://www.music.columbia.edu/pipermail/music-dsp/2003-February/022189.html> (accessed 06/12/2006)

KORTNIK. J, 2006, *J.Kortnik's website* [online] available from <http://web.inter.nl.net/users/J.Kortink/home/articles/sn76489/index.htm> accessed 04/12/2006.

KUBARTH. D., 2006, *Darius Kubarth's SID in-depth information website* [online] available from <http://sid.kubarth.com/> (accessed 06/12/2006)

Leopardcats.com, 2007, *Breadboard pokey in CPLD* [online] available from www.leopardcats.com/bbpokey/breadboard_pokey_project.htm (accessed 28/01/2007)

Mmonoplayer.com, 2007 *Atari_2600 MSP external documentation* [online] available from <http://www.mmonoplayer.com/mspexternals.php> (accessed 17/01/2007)

Sinclair Research Ltd, 1984, *Zx spectrum 48k service manual* [online] available from ftp://ftp.worldofspectrum.org/pub/sinclair/technical-docs/ZXSpectrum48K_ServiceManual.pdf (accessed 21/01/07)

Slack.net, 2007a, *Blarggg Delta modulation*, [online] available from <http://www.slack.net/~ant/nes-emu/dmc/> (accessed 05/01/2007).

Slack.net, 2007b, *Blarggg Saw waves*, [online] available from <http://www.slack.net/~ant/misc/nes-saw/> (accessed 05/01/2007).

Smspower.org, 2006, *SN76489 notes* [online] available from <http://www.smspower.org/maxim/docs/SN76489.txt> accessed 04/12/2006.

TAYLOR. B, 2003b, *Delta modulation controller*, [online] available from <http://nesdev.parodius.com/dmc.txt> (accessed 08/11/2006).

TAYLOR. B., 2003a, *NES Sound*, [online] available from <http://nesdev.parodius.com/NESSOUND.txt> (accessed 08/11/2006).

Texas Instruments, Unknown year, *SN76489 Datasheet* [online] available from <ftp://ftp.whtech.com/datasheets%20&%20manuals/SN76489.pdf>, accessed 04/12/2006.

Tuwien.ac.at, 2007, *Interview with Martin Galway*, [online] available from <http://stud1.tuwien.ac.at/~e9426444/sidtech.html> (accessed 04/01/2007)

Wikibooks.org, 2006, *NES programming* http://en.wikibooks.org/wiki/NES_Programming (accessed 06/12/2006).

Wikipedia.org, 2007a, *Texas instruments SN76489* [online] available from http://en.wikipedia.org/wiki/Texas_Instruments_SN76489 accessed 18/01/2007.

Wikipedia.org, 2007b, *Konami SCC*, [online] available from http://en.wikipedia.org/wiki/Konami_SCC (accessed 20/01/2007)

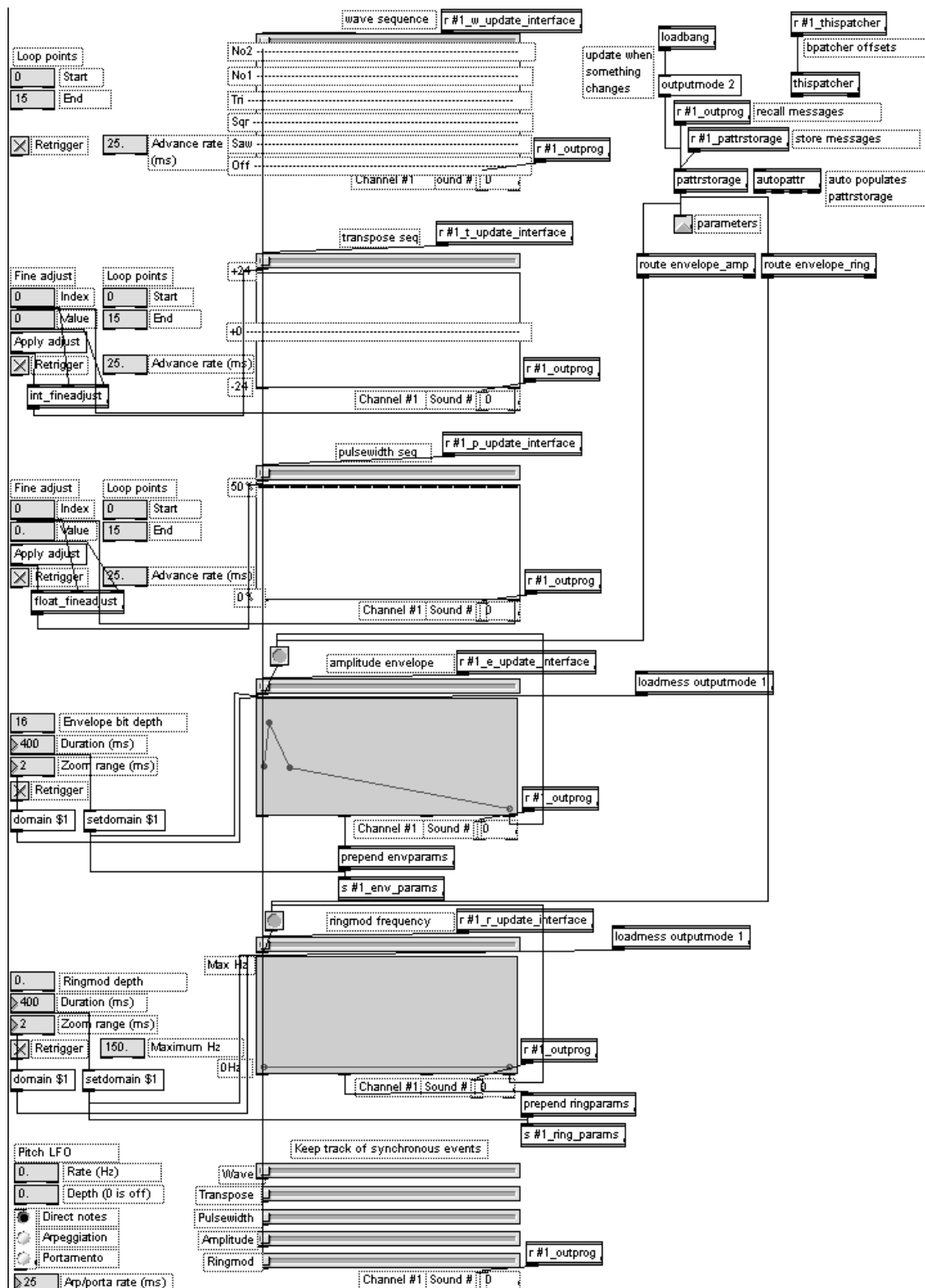
Wikipedia.org, 2007c, *Original Amiga chipset* [online] available from http://en.wikipedia.org/wiki/Original_Amiga_chipset#Paula (accessed 21/01/07).

Wikipedia.org, 2007d, *General instruments AY38912* [online] available from http://en.wikipedia.org/wiki/General_Instruments_AY-3-8912 (accessed 26/12/06).

Yamaha, 1992, *YM2149 datasheet* [online] available from http://mysite.freemove.com/antspants/st_info/psg.zip (accessed 04/01/2007)

Appendix 5 [editor] abstraction full structure

In this diagram, some comment boxes (used here to label controls) and patch cables connecting objects obscure other interface components. This is an unavoidable side effect of showing the full structure. The end user will see an uncluttered display with most of this structure hidden.



Appendix 6

Commented sourcecode for [clockedindex~] external

This section shows the sourcecode of the external developed to provide signal-rate playback of control sequences by advancing through a wavetable at specific intervals specified in milliseconds per sample. It is fully commented by the author.

```
/* clockedindex~.c - modified SDK example of [index~] an object which accesses an
MSP buffer~
*/

#include "ext.h"
#include "z_dsp.h"
#include "buffer.h" // this defines our buffer's data structure and other
goodies

void *clockedindex_class;

// object variables are stored here
typedef struct _clockedindex
{
    t_pxobject l_obj;
    t_symbol *l_sym; // pointer to object arguments in MaxMSP patch
    t_buffer *l_buf; // pointer to buffer
    long l_chan; // number of channels in buffer
    long l_mincount; // loop "start" position
    long l_maxcount; // loop "end" position
    double d_curcount; // current buffer read index position
    long l_holdmsecs; // time before next increment (in milliseconds)
    long l_holdsamps; // time before next increment (in samples)
    long l_sampcount; // current number of samples since last increment
} t_clockedindex;

// the _perform() function is where the signals are processed and output
t_int *clockedindex_perform(t_int *w);

// the _dsp() function is where the object is added to MaxMSP's DSP chain
void clockedindex_dsp(t_clockedindex *x, t_signal **sp);

// the _set() function sets the buffer to access
void clockedindex_set(t_clockedindex *x, t_symbol *s);
// the _new function is called when the object is created in the patch
void *clockedindex_new(t_symbol *s, long chan);

// _inX() functions add inlets (in a right to left order!)
void clockedindex_in3(t_clockedindex *x, long n);
void clockedindex_in2(t_clockedindex *x, long n);
void clockedindex_in1(t_clockedindex *x, long n);

// _assist() is called when the mouse is hovered over the object
void clockedindex_assist(t_clockedindex *x, void *b, long m, long a, char *s);

// _dblclick() is called when the user double clicks on the object
void clockedindex_dblclick(t_clockedindex *x);

// _reset() is called to reset the counters to zero
void clockedindex_reset(t_clockedindex *x);

t_symbol *ps_buffer;

void main(void)
{
    // configures the object to support various types of messages
    setup((t_messlist **) &clockedindex_class, (method)clockedindex_new,
    (method)dsp_free, (short)sizeof(t_clockedindex), 0L,
    A_SYM, 0);
}
```



```

// add response functions to various messages
// these get called when a matching message is sent to the object
// in the 0th inlet
address((method)clockedindex_dsp, "dsp", A_CANT, 0);
address((method)clockedindex_set, "set", A_SYM, 0);
address((method)clockedindex_assist, "assist", A_CANT, 0);
address((method)clockedindex_dblick, "dblick", A_CANT, 0);

// add inlets in a right to left order!
addinx((method)clockedindex_in3,3);
addinx((method)clockedindex_in2,2);
addinx((method)clockedindex_in1,1);

// respond to a reset-to-index-zero in 0th inlet
address((method)clockedindex_reset, "reset", 0);

dsp_initclass();

// set the default buffer to access
ps_buffer = gensym("buffer~");
}

t_int *clockedindex_perform(t_int *w)
{
// the perform function is where samples are calculated and output

t_clockedindex *x = (t_clockedindex *) (w[1]);
t_float *in = (t_float *) (w[2]);
t_float *out = (t_float *) (w[3]);
int n = (int) (w[4]);
t_buffer *b = x->l_buf;
float *tab;
double temp;
double f;
long index,chan,frames,nc;
long saveinuse;

// flag the buffer as "in use"
saveinuse = b->b_inuse;
b->b_inuse = true;

tab = b->b_samples;
chan = x->l_chan;
frames = b->b_frames;
nc = b->b_nchans;

// check if it's time to increment the buffer index position
if (x->l_sampcount >= x->l_holdsamps){

// if we've reached the "loop end" then we should
// wrap around to "loop start"
if (x->d_curcount >= x->l_maxcount){
// set current count to "loop start"
x->d_curcount = x->l_mincount;

// set elapsed sample count to zero
x->l_sampcount = 0;
// otherwise just increment the count normally
}else{
// increment the buffer index position
x->d_curcount++;

// set elapsed sample count to zero
x->l_sampcount = 0;
}
}
}

```

```

// process a complete signal vector
while (n--) {
    *out++ = (float)tab[(long)x->d_curcount]; //output sample
    *in++; // lets be consistent (even though this value is not used)
    x->l_sampcount++; //keep track of outputted samples
}

// un-flag the buffer as "in use"
b->b_inuse = saveinuse;
return w + 5;

// believe it or not, the SDK encourages the use of GOTO statements!
zero:
    while (n--) *out++ = 0.;
out:
    return w + 5;
}

// here's where we set the buffer~ we're going to access
void clockedindex_set(t_clockedindex *x, t_symbol *s)
{
    t_buffer *b;

    x->l_sym = s;
    if ((b = (t_buffer *) (s->s_thing)) && ob_sym(b) == ps_buffer) {
        x->l_buf = b;
    } else {
        error("clockedindex~: no buffer~ %s", s->s_name);
        x->l_buf = 0;
    }
}

// if a "0" (zero) number comes in, this function resets the counter
void clockedindex_reset(t_clockedindex *x)
{
    x->d_curcount = 0;
    x->l_sampcount = 0;
    post("reset curcount to %d",x->d_curcount);
    post("reset sampcount to %d",x->l_sampcount);
} //simple

void clockedindex_in1(t_clockedindex *x, long n)
{
    /* sets holdsamps to appropriate value based on received MILISECONDS

    the holding of buffer index position counter at a value is achieved by
    counting until an elapsed number of SAMPLES have expired, but the user
    specifies the hold time in MILISECONDS. This function performs the conversion
    on the input MILISECOND value.
    */

    x->l_holdsamps = ((float)n/1000.)*44100;
    post("holdsamps now %d",x->l_holdsamps);

    // FIXME
    // *** replace "44100" value with _getSampleRate() function ***
    // what if the samplerate is not 44.1KHZ in a non-standard setup?
}

void clockedindex_in2(t_clockedindex *x, long n)
{
    /* sets mincount to received value
    controls the "loop start" position
    */
    x->l_mincount = n;
    post("mincount now %d",x->l_mincount);
}

void clockedindex_in3(t_clockedindex *x, long n)

```

```

{
    /* sets maxcount to received value
    controls the "loop end" position
    */
    x->l_maxcount = n;
    post("maxcount now %d",x->l_maxcount);
}

void clockedindex_dsp(t_clockedindex *x, t_signal **sp)
{
    clockedindex_set(x,x->l_sym);

    //initialise
    clockedindex_reset(x);

    x->l_maxcount=15;
    x->l_mincount=0;
    post("min %d, max %d",x->l_mincount,x->l_maxcount);

    x->l_holdsamps=44100;
    post("holdsamps now %d",x->l_holdsamps);

    dsp_add(clockedindex_perform, 4, x, sp[0]->s_vec, sp[1]->s_vec, sp[0]->s_n);
    //dsp_add(clockedindex_perform, 4, x, sp[1]->s_vec, sp[0]->s_n);
}

// this lets us double-click on clockedindex~ to open up the buffer~ it references
void clockedindex_dblick(t_clockedindex *x)
{
    t_buffer *b;

    if ((b = (t_buffer *) (x->l_sym->s_thing)) && ob_sym(b) == ps_buffer)
        mess0((t_object *)b,gensym("dblick"));
}

// the information here is out of date, it is left over from the [index~] example
sourcecode
// FIXME: descriptions of all four inlets (0 - 3) should be added here
void clockedindex_assist(t_clockedindex *x, void *b, long m, long a, char *s)
{
    if (m == ASSIST_OUTLET)
        sprintf(s,"(signal) Sample value at Index");
    else {
        switch (a) {
            case 0:    sprintf(s,"(signal) Sample Index"); break;
            case 1:    sprintf(s,"Audio Channel In buffer~"); break;
        }
    }
}

void *clockedindex_new(t_symbol *s)
{
    t_clockedindex *x = (t_clockedindex *)newobject(clockedindex_class);
    dsp_setup((t_pxobject *)x, 1);

    outlet_new((t_object *)x, "signal");
    x->l_sym = s;

    //in a right to left order
    intin((t_object *)x,3);
    intin((t_object *)x,2);
    intin((t_object *)x,1);
    // no need for 0th inlet, get one by default

    clockedindex_in3(x,6);
    clockedindex_in2(x,4);
    clockedindex_in1(x,1000);
    return (x);
}

```

Appendix 7

Time planning - Gantt charts, planning diagrams