Individual Project Module

Project title

An Investigation into Music-Oriented Software-Based Audio Signal Processing, Including Development of a Real-time Audio Application Using C++

Author

Toby Newman

Project tutor

Ahmad Kharaz

Programme of study

BSc (Hons) Music Technology and Audio System Design

Stage 3

University of Derby

April 2002

Acknowledgements and Thanks

Ahmad Kharaz for guidance (Personal tutor)

Steinberg Staff

Members of the Music DSP mailing list

Members of the Steinberg 3rd Party Developers mailing list

Natanya Ford for C++ assistance

Ron Newman

Duncan Werner for initial guidance

Abstract

This study presents an in-depth investigation into the development of a series of audio applications based upon the Steinberg software development kit, detailing the methods used, and effectiveness of results achieved. Development culminates in an advanced ring modulation effect.

A history of sound reproduction within the scope of the personal computer is presented, to allow the context of Steinberg's host based system to be better understood.

By studying computer-based music technology from first principles in Chapter 1, it is hoped that the reader will be more able to follow the more complicated topics covered in Chapter 2.

In conclusion it is found that the host-based system, for which the advanced ring modulation effect is made, is a culmination of specific achievements and methods developed since "Music1", the first music software, written in 1957.

Table of Contents

1 List of Figures or Illustrations

2 Notation

Algorithm
A set of instructions that perform a function or solve a problem

API
Applications Programming Interface.

CPU
Central Processing Unit

Digital Audio Data
A series of discrete values used to store an approximation of an analogue waveform

*.dll file
Dynamic Link Library. Files ending with the extension *.dll contain data and functions that can be accessed by other applications "on-the-fly".

DSP
Digital signal processing – the analysis and manipulation of digital data. Within the scope of this report, DSP is used to refer to digital audio data

EQ
Equalisation is the act of adjusting the sonic properties of a sound by attenuating or boosting the volume of specific frequencies or ranges of frequencies.

FM
"Frequency Modulation", also referred to as 'frequency shift keying'. Modifying the frequency of one signal in proportion to the amplitude of another. (Boone, K. 2000 [g])

GUI
Graphical User Interface.

IBM
International Business Machines. "A multinational company, the largest manufacturer of computers in the world" ("Digital Oracle", 2002 [e])

ISA
"Industry Standard Architecture" is a format of upgrade slot found in home computers.

MIDI
Musical Instrument Digital Interface

“An international standard for communication between a musical instrument and a computer.” (MIDI and Digital Audio Terms, 2001 [h])

Modulation
A method of encoding one signal onto a second “carrier” signal.

Motherboard
This is a large printed circuit board found in most home computers. It includes controller circuitry, and slots for the CPU or PCI and ISA cards.

MSDOS
'Microsoft disk operating system.' The original operating system for IBM compatible PCs.

OPL
Abbreviated name for the Yamaha “Operator tyPe- L” chip used to FM synthesise on early ISA soundcards

O/S
Operating System: This is a piece of software that takes control of computer functions such as accessing disk drives, creating a GUI, and allocating memory. Other programs can then be loaded on top of the O/S and use features offered by it.

Pascal
An object oriented computer language that is very tightly structured to avoid messy code.

PC
Personal Computer; this term was used to refer to the original IBM computers launched in 1981. It can now be taken to mean any modern day computer compatible with IBM machines.

PCI
“Peripheral Component Interconnect”, the successor to ISA, this upgrade slot is faster and smaller.

PIT
Programmable Interval Timer. A chip found commonly on IBM compatible motherboards which can be controlled by software to generate signals at predetermined frequencies.

RAM
Random Access Memory.

Ring Modulation

This is a type of amplitude modulation that creates two side bands whose frequencies are the sum and difference of the input and modulation frequency. It can be used musically to create harmonically complex metallic sounds.

ROM

Read Only Memory.

Shareware

Shareware is a method of licensing software on a "try before you buy" basis, often used by smaller development companies or individuals.

SDK

Software Development Kit; A package distributed to allow 3rd party developers create software for a particular platform.

SID

Sound Interface Device, name of the music chip used in Commodore's C64/128 range of computers

Subtractive synthesis

This is the process of creating a new sound from a source waveform rich in harmonics by the use of filters.

UART

"Universal Asynchronous Receiver Transmitter"; this is the chip in PCs that controls the serial port, a socket that allows communication with external devices.

Virtual Memory

Virtual memory is the simulation of system memory (solid state chips) by use of a computer hard disk. It is considerably slower than system memory, due to its reliance upon mechanical action rather than electromagnetic signals

Wavetable

"Wavetable - A series of numbers stored in memory that, when routed to a Digital-to-Analogue converter, reconstruct a particular waveform. Wavetables can also be used to reconstruct samples of acoustic sounds." (IAEKM, 1998 [i])

While Loop

A C++ programming method of executing an instruction or set of instructions until a given condition is false.

3 <u>Introduction</u>

3.1 <u>Overview</u>

This report is submitted as a final year project for the degree course BSc(Hons) Music Technology and Audio System Design at Derby university. The report has been carried out over the period of 2001/2002.

The issue under investigation is the development of a series of software audio applications to operate under Steinberg's "VST" host-based system.

The VST host system itself is detailed in Chapter 1 as a culmination of developments in computer based music-oriented technology.

The issue's importance lies in the speed at which it is being improved upon. The VST host is a very new technology, and as each new technology in this field is released, development is often already underway on improved systems with which to replace it. As such, there is little time to look back at the origins of contemporary systems, where such a retrospective can yield deeper understanding of current methods.

3.2 <u>Literature Review</u>

When carrying out research into the older areas of computer based music technology it was found that the richest source of background reading was through books such as Electronic Music Synthesis: Concepts, facilities, techniques, *Howe, H.S. Junior.* (1975). Since many of the earlier developments occurred before widespread use of the Internet, it is mainly in these types of volumes that the first steps into this new technology are documented.

The more recent developments in the field of computer-based music technology are best investigated by studying current journals. Articles in *Sound on Sound (SOS Publications Group)* often discuss the more musical aspects of computer-based audio, and games development journals should not be ignored since they make use of many of the current computer-based music technologies.

In order to successfully construct a software application it is important to be aware of protocols and methods by which this is best done. By observing strict development guidelines during early stages, problems caused by complicated applications can be avoided at later stages. An understanding of such practical software methods can be gained by reading *Software Engineering, Ince, D.C., (1994),* a book that details software planning techniques.

While actually writing code it is essential to have some form of C++ reference. "*Object Oriented programming with C++", Parsons D.C. (2000)* was referred to extensively when creating the applications included in this report. This book is recommended for reference since it covers object-oriented concepts, programming, analysis, and design.

3.3 Methods of Investigation

The investigation is primarily intended to demonstrate the creation of a series of software applications, using one of the most modern technologies available to the third party music software developer: the Steinberg VST architecture. This practical element successfully displays the core benefits of many of the new technologies discussed in Chapter 1, where the history of development of computer-based music is documented.

This project report is the culmination of the 3-year Music Technology and Audio System Design course offered at Derby University. It draws from many of the topics covered by the course, but primarily from Audio Applications Programming and Digital Signal Processing.

4 Terms of Reference

4.1 Introduction

This investigation is intended to give an understanding of how current computer-based music technology has come to exist, and highlight specific improvements that this current technology offers. This is done via a realisation of a series of audio processing applications using current technology in chapter 2, in order to demonstrate the improvements documented during the retrospective of computer-based music technology development in Chapter 1.

Limitations

Although technical terms are listed in “2 Notation”, and their usage has avoided where possible, some knowledge of music technology must be assumed due to the technical nature of the report.

Due to the massive array of techniques involved in modern music technology, it has been necessary to make some omissions in order to focus on areas that are deemed most important. This has been avoided where possible.

For deeper understanding of this report it is recommended that background reading be carried out. Texts referred to throughout the report are listed in (13) References and these form a good foundation from which to begin background reading.

4.2 Aims of the Investigation

- Develop a series of working audio applications using C++. This development will culminate in the creation of an advanced ring modulation plugin that offers LFO control over modulation frequency and depth.

- Documentation of the main technological developments to date that have allowed current music-oriented and computer-based audio signal processing technologies to exist.

5 Conduct of the Study

Documents Collected

The study has been conducted by reference to literature available in multiple libraries. Also studied were music technology periodicals, which successfully offered “snapshots” of current technology, at the time of print.

Due to the subject matter in focus, the richest source of information available was through various channels of the Internet.

UseNet proved to be a rich source of information. There exists a large community of individuals, many of which are fanatical about music technology and are keen to divulge information regarding many areas of music technology. UseNet groups such as comp.music.research and comp.dsp offer a wide source of reference and aid.

The World Wide Web allows access to many pages of information regarding both the development of music technology and methods involved in creating applications using contemporary technology.

6 Chapter 1 - Documentation of the technological developments in music-oriented and computer-based audio signal processing to date that have allowed current techniques to be developed

This chapter is intended to give a foundation of knowledge with respect to the history of music development within the scope of computers. By investigating each of the developments in this area that have contributed to current systems, a deeper understanding is formed that will allow the reader to understand the processes detailed in the second chapter.

Computers have become an increasingly common tool in music within the last twenty years. Recent years have seen a rapid development of various software packages and new application possibilities for computers in music related digital signal processing, recently culminating in real-time synthesis and real-time effects.

The evolution of methods by which the home computer has been able to output sound in any practical form can be traced through its various development processes

The scope of this report covers the use of computers as synthesis tools and audio effects units, and the production of music in multimedia applications such as computer entertainment.

6.1.1 "Music1" - The first music software

Early computers (pre 1970) were extremely slow when compared to current machines, and as such, they where incapable of creating any kind of musical sound in real time.

Max Mathews created the first example of music software in 1957 while working for Bell Laboratories. Named "Music1", it worked by running in batches, slowly generating audio data that was spooled onto mass storage such as tape. This data could then be fed to digital-analogue converters (D.A.C.s) to become audible.

The Music1 software was updated to Music2 in 1958, and Music3 in 1960, each update carrying more functions than the last, including programmable digital wavetables, scoring, timbral variation, modularity and orchestration.

When Mathews finished Music4 in the early sixties, he gave it to Princeton and Stanford universities where it was modified by Hubert Howe and Godfrey Winham to allow controllable envelopes; They also gave it a resonant filter command, and as such created the first software based subtractive synthesis system.

Another alteration they made was to adapt it to use less IBM 7094 processing time by using BEFAP assembler, and subsequently named their version "Music4B".

In 1969, the IBM 7094 at Princeton was replaced by a newer machine, which meant that the BEFAP assembly code would no longer run.

As a result, the code was completely revised and designed for the newer machine. Speed has always been important in computer based music technology, and it was no different in 1969. Users of Music360 had to design their sounds using a laborious punch-card interface and there was no way to immediately hear the results of their labours. As such, any increase in speed shortened the trial-and-error sound development cycle and made practical application of the Music program much easier.

It was at this time that the PDP-8 computer was being researched as a potential platform for real-time audio production. It was not until 1973 that Digital Equipment Corporation created the PDP-11 computer, dedicated to music, which was installed at Massachusetts Institute for Technology (M.I.T.)

Subsequently, a new version of Music, "Music11", was written for PDP-11

Since this was a dedicated music machine, using code written in assembler (one of the most efficient computer languages), it was capable of real-time processing. A major breakthrough was the ability for it to be "played" by connecting it to a controller keyboard. Real time control simplified the task of creating sound on the Music platform by allowing the results of adjustments to be immediately heard.

With such a leap forward in functionality Music11 remained the standard for almost ten years - until 1985. At this time, it became clear that brand new microprocessor technology would become realistically affordable soon, and that the very fast but un-portable Music11 would become obsolete.

Max Mathews writes of this development: "On returning to MIT in 1985 it was clear that microprocessors would eventually become the affordable machine power, that un-portable assembler code would lose it's usefulness, and that ANSI C would become the lingua franca."(Boulanger, 2000 [3])

C, a language used to run early versions of UNIX in 1973, became the new portable language of choice. Since these times, C has been augmented to create C++, now one of the most widespread computer languages available and the language used to create the advanced audio applications in Chapter 2 of this report.

"Because C was developed as a tool for driving an operating system, it had certain characteristics such as speed, compactness, and some very low level elements."(Parsons 2000 [1])

6.1.2 Transistor technology brings faster, smaller computers

In 1975, concepts of music "minicomputers" where discussed that would be capable of fast execution, allowing the user to have "immediate feedback, so that a user could hear his music as soon as it was computed" (Howe H.S.1975 [5])

These, at the time, would have cost inordinate amounts of money: "It is possible, even likely, that such a system could probably be assembled for less than $100,000" (Howe H.S.1975 [5])

With improvements in transistor design technology, it became possible for computers to be made smaller, bought affordably and kept in the home, whereas before they where unwieldy devices that would dominate one or more rooms and were only commonly found at universities and research laboratories.

Prior to the present day consumer market dominance of primarily IBM compatibles, there became available a series of alternative types of computer available to the consumer. Commodore's C64 and C128 where among the most commonly found, and these machines carried one of the most advanced computer-based sound generation chips available for many years to come:

6.1.3 SID chips in the Commodore C64/128

In 1981, Robert Yannes began work for Commodore on a microprocessor that could be placed in and controlled by a home computer. "I had worked with synthesizers and I wanted a chip like those in a synthesizer" Robert Yannes (1996)

Within four or five months, the development was complete and the Sound Interface Device chip was ready to be used in Commodore's computers.

As stated in the Commodore 64 Programmer's Reference Guide: "SID provides wide-range, high-resolution control of pitch (frequency), tone color (harmonic content), and dynamics (volume). Specialized control circuitry minimizes software overhead, facilitating use in arcade/home video games and low-cost musical instruments." (Alstrup, 1987 [b])

It is capable of producing three voices at any one time. Each voice can be used independently, for example as a snare sound, kick drum sound, and melody sound. These three voices can also be used in unison allowing complex waves to be constructed.

The actual voices are very limited. Each of the three individually consists of an 8-bit wavetable based waveform generator, the output of which can be amplitude modulated using a programmable envelope.

The waveform generator creates four signals (triangular, sawtooth, variable pulse-width waves, or noise). The mixing of these signals together allows timbre of the output to be controlled.

Using subtractive synthesis the signal can be altered further. This was done using the programmable filter included in the SID hardware, which had a cut-off range of between 30 Hz and 12 kHz, with a 12dB/octave roll-off. It was selectable between low pass, band-pass, high pass and notch, and had variable Resonance

The chip could also accept audio input and process that using the same techniques described above. This allowed for multiple SID chips to be arranged in sequence to create more complex sounds.

The result was a very electronic sound, but at that time, the sound of the SID chip proved to be a leap forward in imitation of real instruments at a scale that was affordable by the consumer.

The success was partially due to its microprocessor-controlled frequency selection, allowing precise frequency adjustments. This opens up possibilities of using techniques such as slight detuning of voices to create phasing effects.

The new microprocessor-controlled envelope function also allowed closer mimicking of real instruments than that which had previously been possible in real time on such a small scale.

International Business Machines (IBM)

Many of the equivalent computers available at the time of the Commodore C64/128 had similar sound generation chips but none offered the sharpness of tone or frequency accuracy of the SID chip. As such, this was the most advanced home computer-based music device available for some time, until the IBM compatible, began to develop from a mostly text-based machine into the multimedia devices common in offices and homes today.

6.1.4 The PC Speaker

Developed circa 1981, The PC Speaker was the first method by which the IBM compatible home computer issued sound. It was used to create music and primitive sound effects (such as rising pulse sweeps) and to emit beeps of acknowledgement to the actions of the user in other applications.

The PC Speaker itself is simply a small drive unit attached to the inside of a computer case.

It is usually wired directly to a jumper connection on the computer's motherboard, although some soundcards such as the Terratec EWS64XL offer the alternative to transfer the signal from the motherboard to monitor s or Hi-fi speakers.

Sound output of the PC Speaker "is controlled by the Programmable Peripheral Interface device 8255A© and the Programmable Interval Timer 8253© chips." (Hilderink, G.H. 1998 [f])

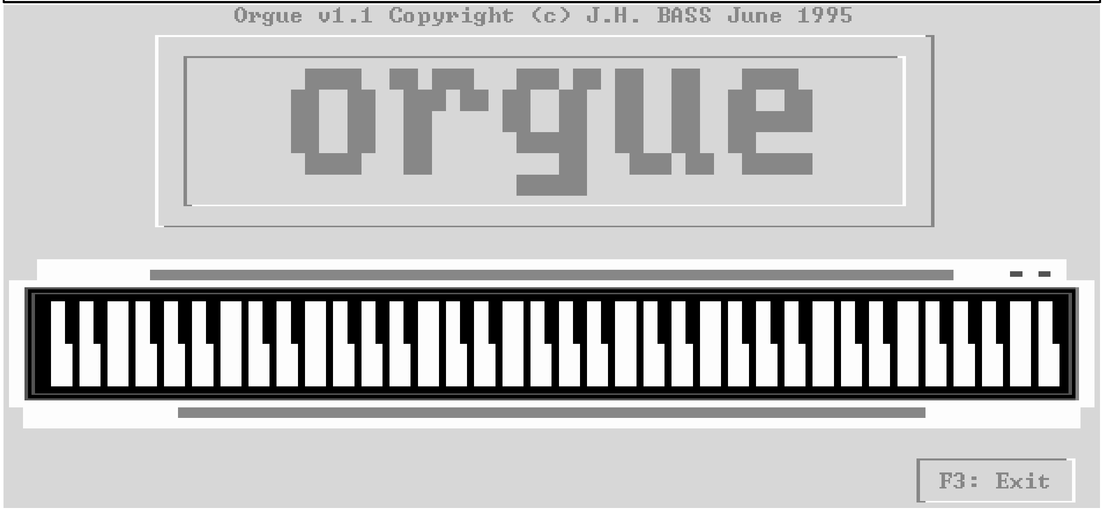
The sound is produced very simply; Channel 2 (port 42h) of the 8255A chip on the motherboard is connected to the computer's speaker and issues "on/off" signals (i.e. square wave pulses) to create sounds. By altering the interval of the Programmable Interval Timer (PIT) chip, the programmer can change the sound frequency outputted at channel 2.

In order to simulate the effect of a chord, it was necessary to quickly play the single notes of the desired chord in a rapid arpeggio, creating the illusion of polyphony.

This method of producing sound has very low functionality. The user may only set the output frequency, and then turn the speaker on/off at that frequency. As such, the PC speaker was totally unsatisfactory for reproducing any form of realistic sound, and was restricted to electronic "beeps". Since the development of better technologies, the PC Speaker has been disregarded for all sound applications with the exception of reporting critical system errors where all other more complex sound reproduction techniques cannot be relied upon to function correctly.

As the first step in creating sound from an IBM compatible personal computer, the PC speaker must not be ignored. However, as new technologies where developed, sound quality from the IBM platform was to increase dramatically. With regard to user-friendliness, an individual wishing to create music using the PC speaker would either have to code a solution themselves or use limited applications with little functionality and no polyphony such as “orgue.exe” created by J.H. Bass (1995, published at www.simtel.net) as seen here.

Figure 6-1 Screenshot from "Orgue.exe", a PC-Speaker music application



This application allows the user to play the PC speaker using the QWERTY keyboard but offers no timbre control or recording facility. There was very little software available which allowed the user to actually sequence musical recordings to be generated from within the computer itself, since the sound quality was so low that few people would have desired to do so at that time.

6.1.5 Soundcards

6.1.5.1 Adlib Music Synthesiser

The next major improvement in computer-based audio technology, the “Adlib Music Synthesiser” became available in 1987 as an ISA card that could be plugged into IBM computers as an upgrade. This upgrade card was the first step away from the basic PC speaker sounds, and used the Yamaha YM3526 FM operator type- L (OPL) chip to generate an FM synthesis wave pattern to recreate basic instrument sounds.

Polyphony was split between 9 melody voices or a combination of six melody voices and five rhythm voices. The FM sound generator was actually the same as the one used in the Yamaha DX-7 synthesizer, employing both a synthesizer and a noise generator, used to create five percussion sounds (bass drum, snare drum, high-hat cymbals, top cymbal, and tom-tom).

The synthesizer allowed sound envelopes to be specified, and offered low frequency oscillators to control vibrato and amplitude modulation.

Adlib became the most popular ISA soundcard available for home computing. The music created by computers and basic sequencers became greatly improved with a departure from esoteric bleeps and buzzes and a valid attempt at real instrument emulation.

The OPL chip became so popular that it was soon available across the majority of ISA soundcards, being added to Creative's market leading "SoundBlaster" series of soundcards in 1989.

This early OPL chip (YM3526) should not be confused with OPL2 (YM3812) or OPL3 (YMF262), both of which are developments of the OPL and offer greater functionality.

OPL2 was the successor to the OPL chip, and was used on most "SoundBlaster" 1.x and 2.x cards.

It has an improved maximum of two frequency modulation operators per voice and allows for up to nine simultaneous voices. With just two FM operators per voice, the OPL2's timbres are still far from realistic.

To follow was the OPL3 chip. This device was still an FM synthesiser, but its increased 4 operators per voice resulted in a less electronic and artificial sound.

This OPL3 could also support up to 18 voices and generate stereo tones of up to 44.1kHz.

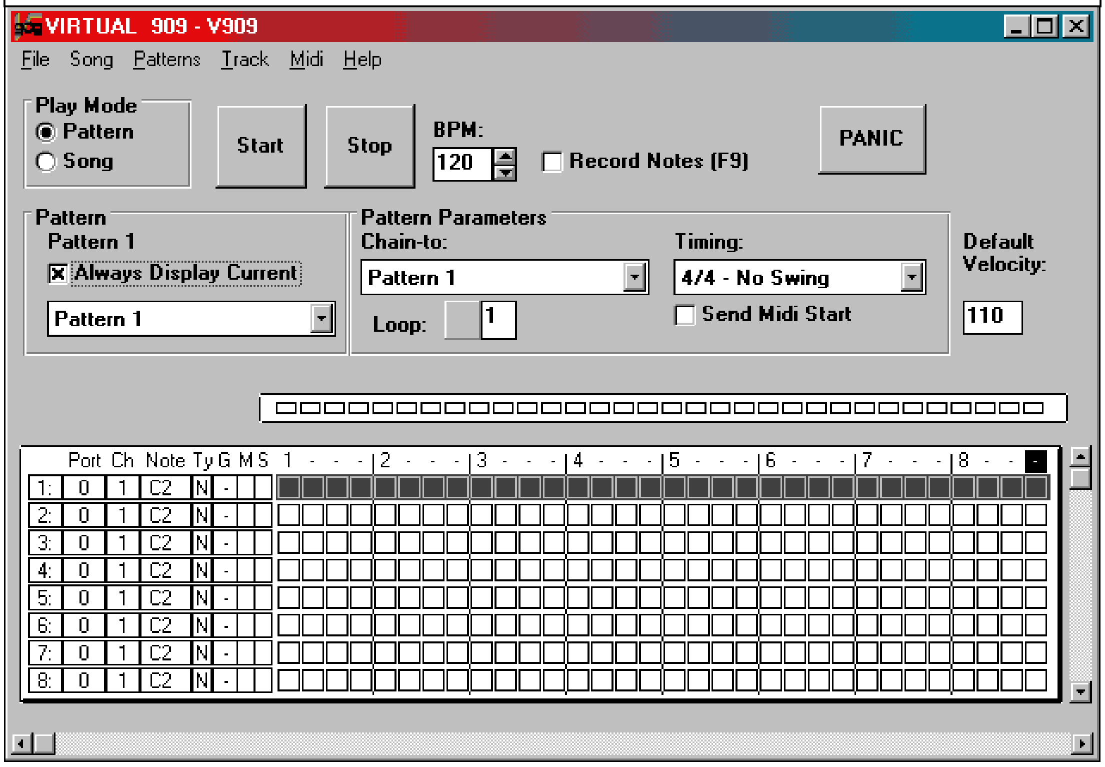
6.1.5.2 Computer-based music sequencer software packages

The features of the OPL chips, coupled with the release of Windows 3.0 in May 1990, beckoned the rise of computer-based MIDI sequencer software packages, the forerunners of today's virtual studio environments.

Windows 3.0, a graphical user interface designed by Microsoft to augment MSDOS, has the ability to access memory beyond 640K which allowed much more powerful software to be run on home computers, where before memory was a crippling restriction for developers.

With the instruments supplied by the OPL driven soundcards, and the possibilities given by the new operating system Windows 3.0, these sequencing applications allowed home users to create songs entirely on their PCs, with the possibilities of percussion, lead instruments, bass and many other timbres offered by the OPL FM synthesisers.

Figure 6-2 A screenshot from V909



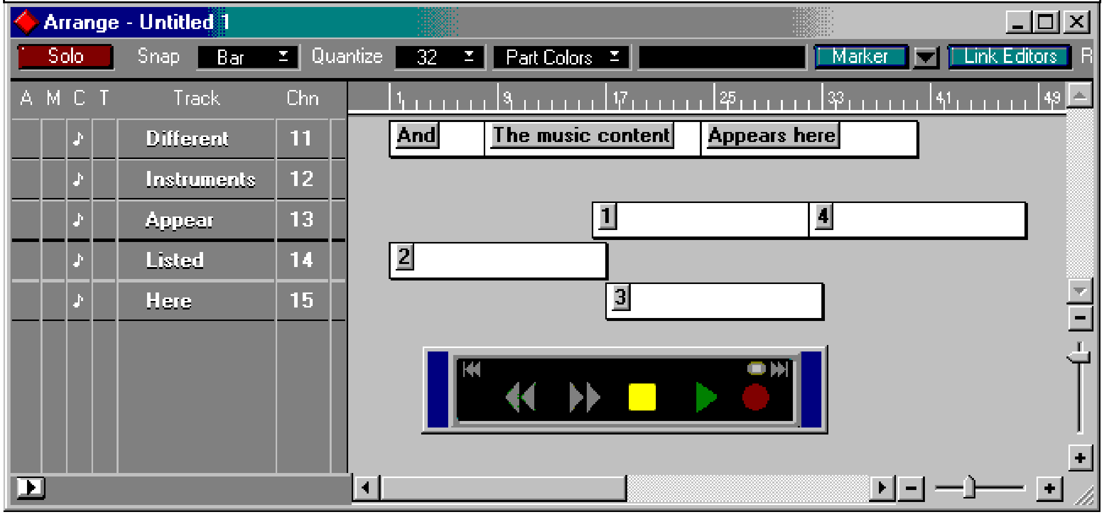
Early sequencer software was very primitive by today's standards. One example of an early sequencer is V909, by Christopher List, shown here.

This MIDI sequencer has no digital audio capability, and the pattern oriented user interface is less intuitive than modern timeline-oriented sequencers.

This awkward user interface is typical of many of the sequencers available at the early 1990s, and is similar to the interfaces offered by the Tracker software discussed later in this report. (see Chapter 6.1.5.4, Page 18)

As synthesis methods improved over time, sequencers began to adapt to take advantage of multitimbrality. The layout was modified from the mathematical tables of the *.MOD and grids of the V909 pattern sequencer, and became something closer to traditional music score. With newer sequencers, time is represented horizontally, and the vertical axis is divided up to denote different instruments, or timbres on a synthesiser.

Figure 6-3: An example of a contemporary sequencer interface, Steinberg's Cubase VST



Derivatives of this design are found in the majority of contemporary sequencer packages.

6.1.5.3 Roland MT-32

An alternative technology that was available at the time of the OPL chips was Roland's MT-32 synthesiser.

This was not a soundcard but an external box that connected to the computer via a "Musical Processing Unit, model 401" (MPU-401) ISA interface card, also designed by Roland.

The MPU-401

This MPU-401 card was the first MIDI interface designed for a computer.

It carried many features such as a built in metronome, and a hardware tape-synch jack to allow the connection and synchronisation of a magnetic analogue tape recorder.

It became such a widely used MIDI interface that Roland established a standard that many future soundcard companies would follow. The MPU-401 is now available as a single chip rather than a whole ISA card, and it is very common to find it included on contemporary soundcards. These chips are often marked as operating in UART mode, which means that they offer the midi interface functions without all the intelligent features such as built in metronome and tape-synch.

An example of such a contemporary card is the Terratec EWS64XL, described as having "Two MPU-401© compatible MIDI interfaces (UART mode)" (Terratec 1998[7])

Linear Arithmetic (LA) synthesis

The MT-32 synthesiser, that was designed to connect to this MPU-401 MIDI interface, operated using Linear Arithmetic (LA) synthesis.

As a pre-cursor to the wavetable synthesis methods used today, LA synthesis was a combination of traditional additive synthesis and basic elements of wavetable synthesis. (See Section 6.1.6)

Sample lengths where heavily restricted at the time of the MT-32, since memory was an expensive commodity, and digital audio samples are often very large, particularly by the standards of the early IBM compatibles. (10 seconds of stereo 44kHz digital audio data takes up 1,764,000 bytes of memory.) These early machines had just one megabyte of memory, of which 354K was assigned to system tasks, leaving only 640K of memory available for applications. This meant that there was no room in system RAM to store lengthy audio samples.

To avoid this problem, a compromise was made between realism and memory usage:

LA Synthesis used very short samples of just the attack elements of instruments, and then augmented them with additive synthesised sustain and release sounds. This allowed for greater accuracy in timbre modelling than anything that had been achieved before using additive, subtractive or frequency modulated (FM) synthesis.

Since only the attack portion was sampled, it required much less storage, allowing for hardware that came installed with less memory, greatly reducing production costs.

6.1.5.4 Increased storage ability leads to more widespread use of digital audio

As each year passed, computers became faster, and digital storage such as RAM and hard disks became larger and dramatically cheaper. Soon, it became viable for short sections of digital audio to be stored in virtual memory on hard disks, or even loaded into system RAM.

SoundBlaster

Since digital audio could now be stored within computers, the manufacturers of soundcards began including Digital to Analogue converters (D.A.C.s) on their products.

The first soundcard generally available to offer this capability was Creative's SoundBlaster. This card also came with an Analogue to Digital converter (A.D.C.) allowing line-level signals to be recorded to the hard disk. These recorded sounds would be stored in *.WAV files, a standard type of file created by the Microsoft Windows 3.0 operating system. As such, the SoundBlaster allowed the user to record and playback any real sound.

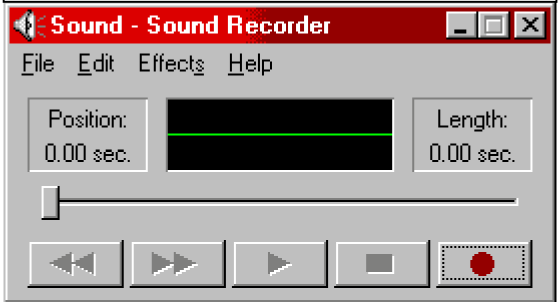
The quality was poor by today's standards. Samples where taken at only 8-bit and any sample rates above 22kHz would have taken up too much storage space when hard drives where commonly between 50 and 300mb.

Once stored on the hard disk, digital audio could then be streamed using driver software through the D.A.C. and out of the computer as an analogue signal ready to be fed to an amplifier and speaker arrangement.

Front-end applications could then offer directory-browsing features with which to find audio files; the software would then operate by reading the stored digital audio data from the hard disk or memory, and communicate that audio to the soundcard using driver software, to produce sound. The early audio applications often included simple controls such as play, pause and stop, to facilitate the playback of audio files.

An example of early digital audio playback software is SNDREC.EXE, a simple program that was bundled with Microsoft's Windows 3.1. This software operated by loading digital audio from the hard disk into memory and then playing back said audio. It was also capable of receiving digital audio from a soundcard's A.D.C. and storing that data on the hard disk in *.WAV format, as such, using the computer as a primitive sampler.

Figure 6-4 A screenshot from SNDREC.EXE



MOD Format

This low fidelity sampling capability spawned the creation of new software by third parties that allowed the samples to be both played back and sequenced or mixed. This software was termed the "Tracker", and the primary file format that was used was the "*.MOD".

Tracker songs come in the form of one inclusive file which is played back using tracker software; that file includes both timing information like a MIDI file, and embedded digital audio sample information.

"Modules are digital music files, made up of a set of samples (the instruments) and sequencing information, telling a mod player when to play which sample on which track at what pitch, optionally performing an effect like vibrato, for example. Thus mods are different from pure sample files such as WAV or AU, which contain no sequencing information, and MIDI files, which do not include any custom samples/instruments." (Defacto2, 2001 [c])

Much of the software written to create *.MOD files was coded by enthusiasts and distributed as shareware. As such, it lacked any standardisation, and the MOD file came in many sub-formats, such as *.IT, *.MOD and *.S3M.

This new *.MOD software sampling technology allowed improvements in sound quality coupled with relatively small file sizes. The file sizes were kept small because whole songs did not need to be stored as a continuous *.WAV file; repeating sections (such as drum loops) could be simply stored within the *.MOD file as a single loop, and then “called” by the sequencing software to play when needed.

These *.MOD files had numerous benefits over MIDI files. Any cheap D.A.C. enabled soundcard would suffice to play the files back, since the sound quality was almost entirely based on the software. This is unlike MIDI files where the quality is dependant upon the MIDI playback hardware. As such, a MIDI file will often sound quite different when moved from one computer to another, whereas a *.MOD file will always sound identical.

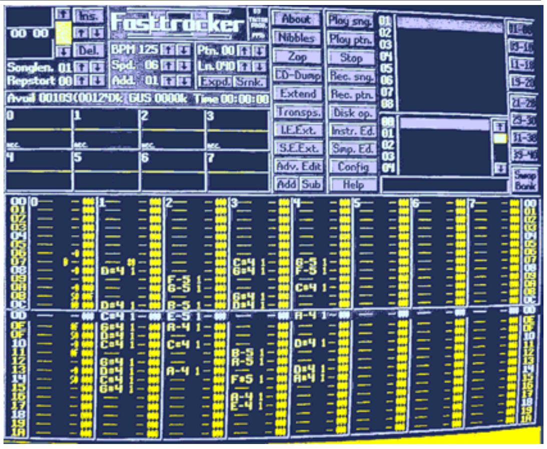
Due to the nature of the *.MOD, whereby up to 16 audio samples had to be mixed down onto one or two 8-bit hardware output channels, there was always a loss of quality. Older *.MOD files are characteristic for their “grainy” timbre, caused by bit quantisation at this mixing stage.

The *.S3M format, a development on from the *.MOD format, permitted the use of samples up to 44kHz, but due to hardware available at the time, the output was still restricted to 8 bits.

Soon, a new piece of hardware, the Gravis Ultrasound, was developed which allowed up to 32 dedicated digital audio output channels. This neglected the need for lossy mixdown to one or two 8-bit audio channels and allowed the *.MOD format to play back at much higher quality. The Gravis Ultrasound was a very popular card among musicians for this reason.

Writing music in tracker format is very difficult. Unlike modern day sequencers, which allow traditional musicians to compose music using scores, the tracker file was created using tables of numbers and often played using the QWERTY keyboard rather than a traditional piano keyboard.

Figure 6-5 Screenshot from "FastTrackerII"

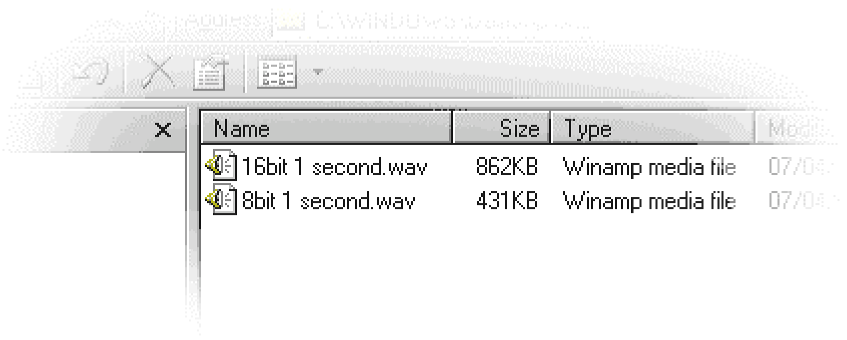


An example of a popular tracker program, "FastTrackerII", is shown here. The layout of this software is comparable to many other tracker programs. The digital audio is loaded from the hard disk into the storage area at the top right of the screen, and the table that dominates the lower half of the screen contains timing and pitch information for playing back the samples that have been loaded.

6.1.5.5 Digital Audio becomes 16 bit

16-bit audio takes up more storage space as 8-bit audio. This can be shown by creating two one-second audio files, one at 16 bit and the other at 8 bit. By saving both, the size can be easily compared using Microsoft's Windows Explorer.

Figure 6-6 Size comparison of two identical-length 8 and 16 bit audio files



Name	Size	Type
16bit 1 second.wav	862KB	Winamp media file
8bit 1 second.wav	431KB	Winamp media file

As seen here, the 16-bit audio file is twice the size of the 8-bit file.

Since there is a x2 increase in size, computer storage needed to grow in capability before 16 bit samples of any usable length could be stored. Fortunately, storage technology improves very quickly:

"The increase in data storage capacity of hard disk drives (HDDs) has increased at roughly the same rate as the speed of microprocessors, with a 60% compound annual growth rate." (Gorham Consulting [j])

The SoundBlaster 16

With this rapid increase in size, it was in 1992 that Creative saw a market for their SoundBlaster 16, a card with similar features to the original SoundBlaster but offering 16-bit audio and the updated OPL3 FM synthesiser.

As users began to buy the newer SoundBlaster cards, software was written that took advantage of the 16-bit capabilities.

Also, the tracker format was updated to make use of 16-bit audio files; these new tracker files had the extension *.XM.

The *.XM format also offered other features such as volume / pitch envelopes and multi-sampled instruments.

Multisampling

In early tracker software, when a sample was loaded into memory, it could be played back at any pitch. This allowed musical sounds to be spread across the keyboard and played like a piano. However, the further away from the "root pitch" (the pitch where the sample was neither sped up nor slowed down) the less realistic the sound became. This is because by speeding up or slowing down musical samples, the apparent resonant frequencies of the instrument that is being copied become less realistic. As such, the higher the pitch of the sample, the smaller the instrument sounds, but the lower the pitch, the larger the instrument sounds.

A second drawback of altering the pitch of a sample is that the original sample has its sample rate fixed when recorded. If, for example, a sample is recorded at 22kHz, then increasing its pitch will necessitate an increase in this sample rate. In order to play this re-pitched sample through 22kHz output hardware, it will have to be re-sampled down to 22kHz again by interpolation. Linearly interpolating a sample's value based on the two closest samples has the effect of introducing harmonic distortion.

Multisampling allows several samples to be used to represent one instrument. One sample, for example, could be used to represent each octave of the instrument. This dramatically reduces the amount of pitch modification necessary, reducing harmonic distortion, and creating a more natural timbre.

6.1.6 <u>Wavetable Synthesis</u>

Wavetable synthesis is an extension of the multisampling techniques of the tracker.

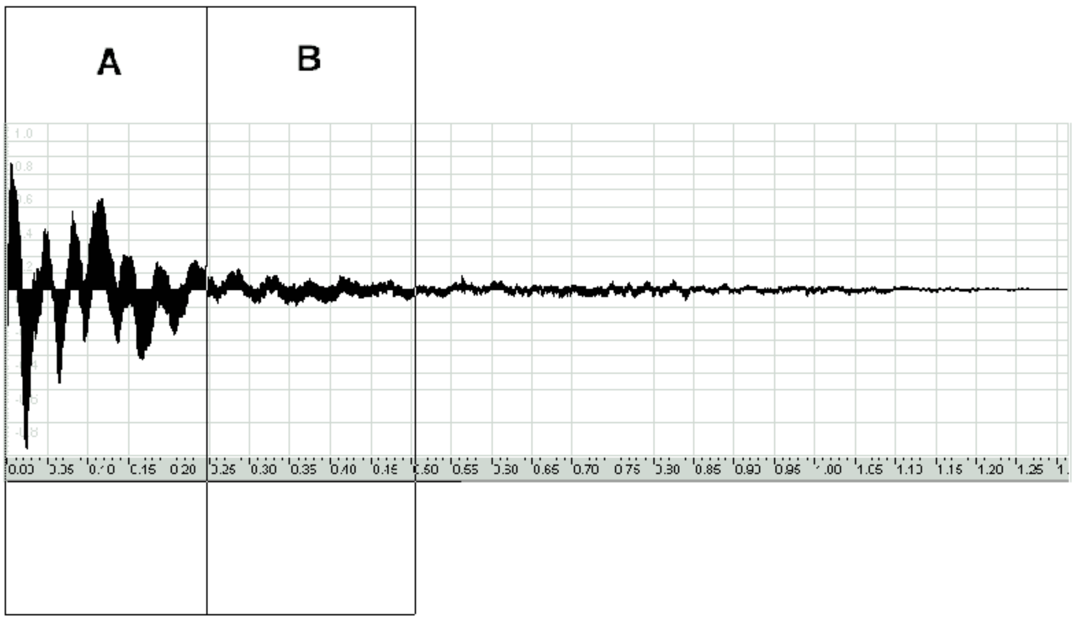
"In Wave Table synthesis a desired pitch is achieved by taking a pre-recorded sample of one pitch and playing back the recording faster or slower to match the desired pitch" (Net Express, 1996 [k])

Whereas with the tracker it was only possible to play samples as an instrument from within the tracker software, companies developed hardware that offered the same possibilities that could be used alongside any chosen sequencer software.

The Gravis Ultrasound (GF1) is one example of wave table synthesizer. It has RAM onto which instruments can be loaded and played back using MIDI messages.

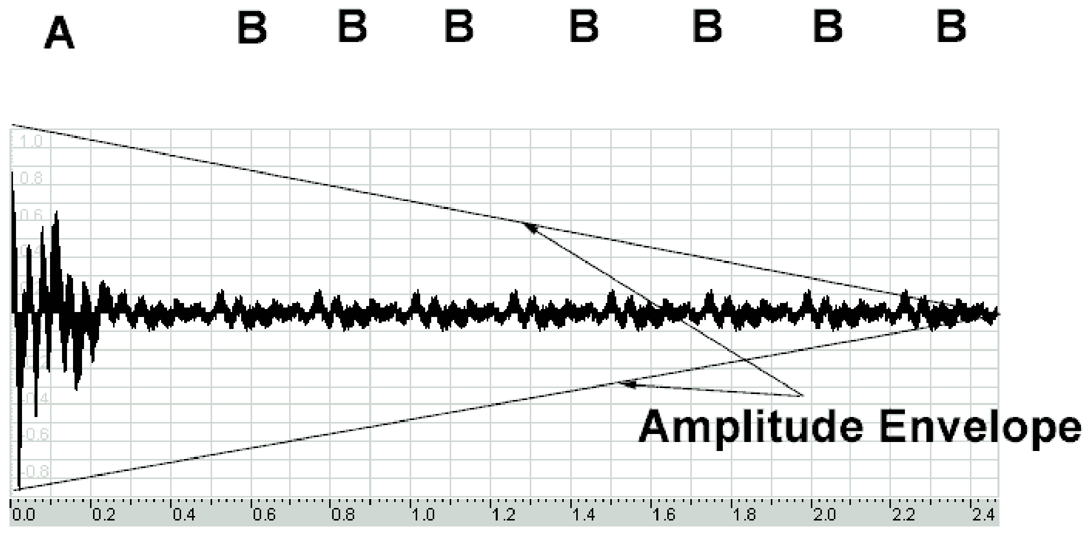
Creative also released a revision of their SoundBlaster16, the SoundBlaster16/ASP, that could accept a "Wave Blaster" daughter board. This daughter board was a separate piece of hardware that could be plugged into a slot on the SoundBlaster16/ASP, giving the added functionality of wavetable synthesis. The instruments where made of 16-bit recorded samples, and could be played back in stereo. Sound quality was restricted due to the onboard ROM that stored the samples. At only 1Mb in size, samples had to be kept very small. It was common for looping to be used in order to save storage space. Rather than storing complete instrument samples that often include sustained decays, only the attack (A, below) and first part of the instrument's decay (B, below) were sampled.

Figure 6-7, only parts A and B were kept from the original sample



When the instrument was played back, the attack portion (A) was played once, and then the portion of the decay (B) was looped as an amplitude envelope faded it out. This gave a good approximation of the original sound, but also saved a great amount of storage space.

Figure 6-8 Part "B" was looped and faded, as shown here



Creative's SoundBlaster series of cards remained the standard for computer-based sound reproduction technology for many years after the introduction of the SoundBlaster16. Subsequent versions of the card differed very little in the core technology they offered.

The SoundBlaster 32 was in most aspects identical to the SoundBlaster 16 except that it offered 32 wavetable voices, and the SoundBlaster 64 that followed offered 64 simultaneous voices. One new capability that was introduced with later versions of the SoundBlaster 32 was the inclusion of up to 28Mb of RAM. Users could record their own samples to the hard disk using the card's A.D.C., and upload them to this RAM to be played as multisampled instruments.

Creative's SoundBlaster 128 increased polyphony again to 128 instruments, and also came with an EMU8000 ASIC chip manufactured by the hardware sampling company E-mu.

Where previously linear interpolation had been used to alter the pitch of samples, thus introducing harmonic distortion, this new chip allowed four data points (four individual samples in series) to be interpolated as a curve using polynomials. This greatly increases the accuracy of the re-pitched sound and reduces harmonic distortion. 32-bit mathematics is carried out inside the chip in order to derive the final interpolated sample.

Multitasking and the audio software suite

Microsoft's Windows operating system had allowed "multitasking", the ability to run more than one application and switch between applications, since the release of version 3.0. Due to the size of digital audio files and thus the hardware-intensive nature of audio software, it did not become practical to multitask audio applications until computers had increased dramatically in speed. In 1995, when Microsoft released Windows95, software developers began to take advantage of this multitasking capability to create audio applications that where designed to run side-by-side as suites.

6.1.7 Mellosoftron and the Software Wavetable.

The first digital audio oriented use of this multitasking technology was the software wavetable. This was emulation in software of the wavetable features offered by the soundcards of the early 1990s. Sound samples where stored on the hard disk, and loaded into the computer's system RAM rather than dedicated soundcard's RAM. From here, a software application could be used to re-pitch the samples and then communicate the sound using driver software to the D.A.C. of the soundcard for playback.

One of the more popular software wavetable applications available was Mellosoftron, written by Polyhedric Software.

This application could be multitasked alongside a MIDI sequencer application. The user would compose music on the sequencer to be played by the Mellosoftron, and at playback the use of MIDI driver software would send the MIDI instructions from the sequencer to the Mellosoftron.

This technique of storing the sounds in system memory meant that the restrictions of sample length were loosened. Between 1995 and 1999, computers commonly carried from 4Mb up to 32Mb of system memory. This was a great improvement over the 1Mb available on soundcards at that time.

Drawbacks

Of course, some of this memory was needed for system operations such as the O/S and applications. By claiming large portions of system memory for the storage of samples, it was not uncommon for the computer to run out of RAM and be forced to use virtual memory instead. This had the effect of slowing the computer down dramatically, and often caused timing problems and glitches in the audio playback.

A second problem was that of CPU load. The relatively slow CPUs of that era (often between 66MHz and 160MHz) had difficulty completing all the tasks demanded of them at sufficient speed to facilitate smooth audio playback. Machines at the slower end of the scale would glitch during playback, and could often lock up completely when asked to run a sequencer, a software wavetable, and an O/S simultaneously.

Direct Music

Microsoft has included a software wavetable in their DirectX windows extensions since version 7.0. Termed "Direct Music", it is an API that allows programs to play music using a generic MIDI sound-set that is identical on every DirectX enabled computer. This gives standardisation in timbre between computers.

Since it inclines game developers towards once again generating music on the fly using MIDI rather than continuing to rely upon Redbook CD audio, it is going some way towards re-instating the dynamic music techniques popularised by the MOD format.

"DirectMusic … offers consistent playback of sound sets using an open standard, Downloadable Sounds Level 1 (DLS1). On top of that, DirectMusic opens more than one door to achieving adaptive musical scores in games". (Hays, T. 1998 [3])

6.1.8 Virtual Synthesisers

6.1.8.1 Offline Virtual Synthesis

Where Mellosoftron had previously emulated a wavetable synthesiser, in 1994 developers took this idea and began to emulate complete hardware synthesisers using the power of newer computer CPUs.

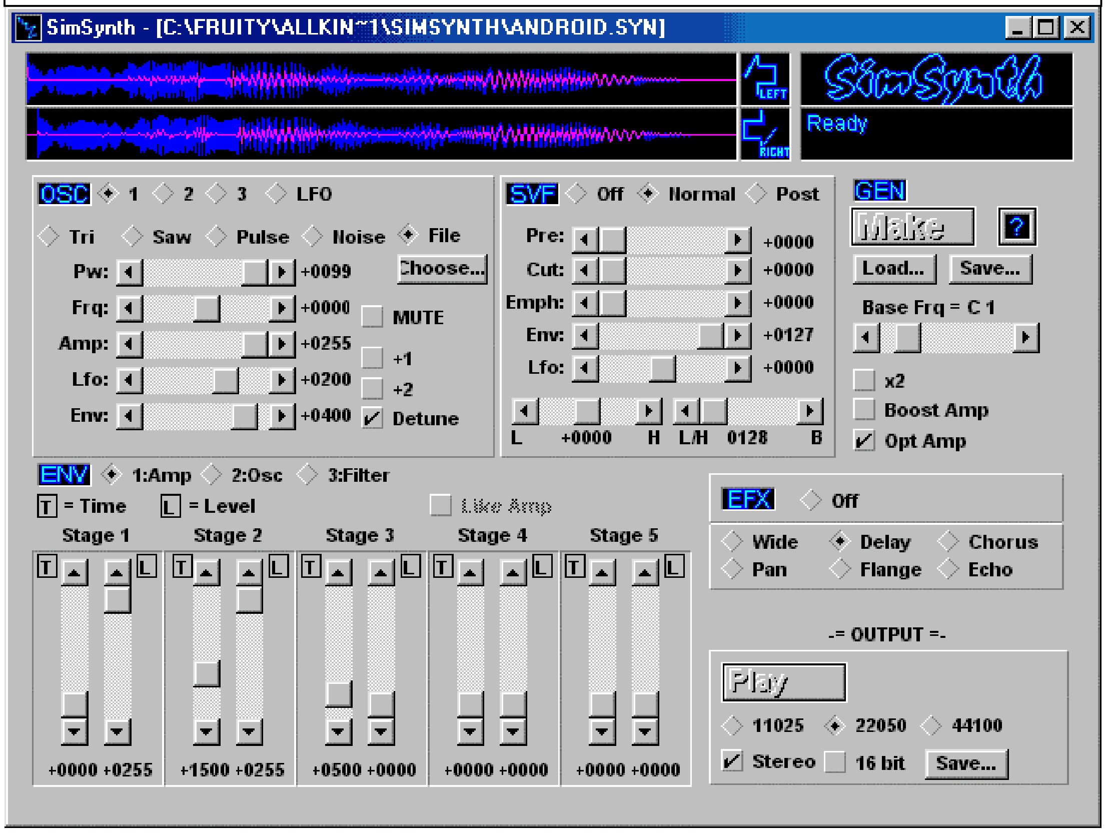
Initially these programs ran offline, not in real-time. The user would be presented with an array of sliders on screen that altered variables in the software. These sliders were similar in function to those found on traditional analogue sequencers, controlling for example oscillator waveforms, filter cut-offs, oscillator pitches and envelope times.

Once the user was happy with the slider's arrangement, the software would be instructed to generate a *.WAV file on the hard disk. This *.WAV file, usually containing a single musical note, could then be loaded from the hard disk onto any wavetable and used to create music.

SimSynth

Released in 1994, SimSynth 1.0 was the first generally released software synthesiser.

Figure 6-9, Screenshot showing SimSynth's controls



SimSynth - [C:\FRUITY\ALLKIN~1\SIMSYNTH\ANDROID.SYN]

Ready

OSC [✓ 1] 2 3 LFO

Tri Saw Pulse Noise [✓ File]

Pw: +0099
Frq: +0000
Amp: +0255
Lfo: +0200
Env: +0400

MUTE +1 +2 [✓ Detune]

SVF Off [✓ Normal] Post

Pre: +0000
Cut: +0000
Emph: +0000
Env: +0127
Lfo: +0000

L +0000 H L/H 0128 B

GEN

Base Frq = C 1

x2 Boost Amp [✓ Opt Amp]

ENV [✓ 1:Amp] 2:Osc 3:Filter

T = Time L = Level Like Amp

Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
+0000 +0255	+1500 +0255	+0500 +0000	+0000 +0000	+0000 +0000

EFX Off

Wide [✓ Delay] Chorus
Pan Flange Echo

-= OUTPUT =-

11025 [✓ 22050] 44100

[✓ Stereo] 16 bit

Due to its offline nature, it was capable of outputting audio files at 44kHz and 16 bits. The interface was divided up into 5 main sections, correlating with the 5 elements to its synthesis model.

The oscillator section allowed the user to choose between a triangle, sawtooth, pulse or noise waveform. This source could then be sent to the filter. An envelope can control both the oscillator and the filter. Finally, a choice of 6 simple effects could be applied to the output before being generated on the hard disk.

Drawbacks of the offline model

SimSynth 1.0 offered a cheap way for musicians to synthesise their own sounds, where hardware to create equivalent sounds could cost up to 1000 pounds.

However, much of the pleasure of analogue synthesis was lost with SimSynth, since without real-time control users where forced to silently adjust arbitrary knobs in the hope that they would produce a pleasing sound upon generation. This was in some ways a step back to the protracted methods of the first offline music software, “Music1”.

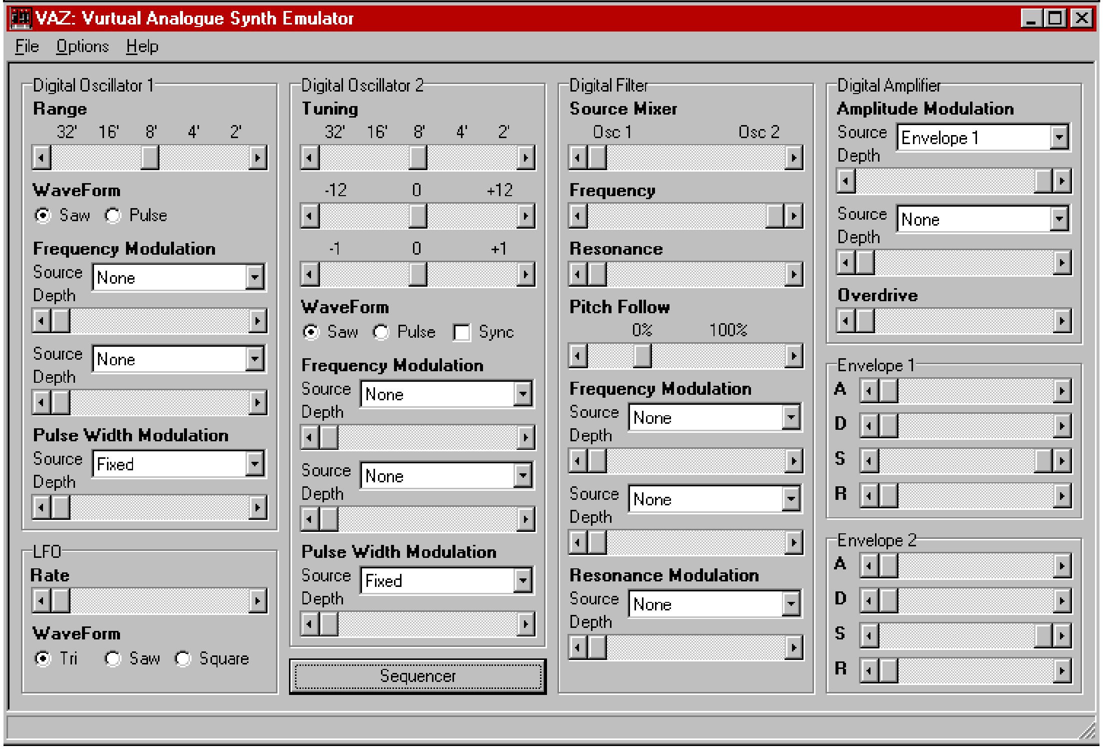
6.1.8.2 Real-Time Virtual Synthesis

As CPUs increased in speed, computers became capable of generating audio “on the fly”. It was possible for software to generate and stream audio data directly to the soundcard with no need for temporary storage on the hard disk. This increase in speed revolutionised much of the computer-based audio signal processing technology available at the time.

VAZ

In 1996, Martin Fay released a program called “VAZ” (Vurtual Analogue Synthesiser) V1.0.

Figure 6-10 Screenshot showing VAZ's controls



This piece of software was very similar in features to SimSynth, but ran in real time on computers with CPUs of 66MHz or faster. Running in real time allowed the user to adjust parameters of the sound while listening to the results, a breakthrough for the IBM compatible platform of similar significance to the real-time capabilities of Music11 in 1973.

6.1.9 Host – Based systems

Currently, general-purpose home computers are powerful enough to cope with a many aspects of musical synthesis and audio processing. The majority of both IBM compatible computers built today have high quality audio output capability pre-installed as standard, in the form of PCI soundcards, or sound cards pre-built onto the system's motherboard.

The software available for these machines has continued to be improved. The new standard for home-computer music technology is the "Host-based System".

Software synthesisers such as VAZ were standalone applications that had to be executed alongside sequencer software. This carried the processing overhead of maintaining two complete applications in memory at one time, with no sharing of resources. The breakthrough in computer-based audio software that is the host-based system allows these types of software to run from within sequencer applications.

This system is utilised by installing one of many "hosts" available on the market today. These hosts are similar in design to the midi and audio sequencers that have been available since the early 1990s but carry new capabilities that allow the end user to install 3rd party "plugins" in order to augment the capabilities of the host.

A plugin is a piece of software that is opened from within the host and shares some of the host's resources. This reduces memory and CPU overheads and allows for more complex audio software to be run than in previous multi-tasked arrangements.

Many of the plugins available model traditional analogue effects units such as EQs and vocoders, where others use DSP to create newer digital effects such as pitch shifting.

Plugin technology is now adopting many of the ideas discussed throughout this report and making them it’s own. Current versions of software wavetables such as Mellosoftron are now run from within VST hosts.

Software synthesisers, too, have been adopted by the host-based system. Native Instruments have released FM7, a plugin version of the Yamaha DX-7 FM synthesiser, which shares it’s sound generator with Yamaha OPL chips discussed earlier in this report.

As can be seen, many of the plugins available for the VST system offer simulation of traditional hardware methods by the use of software emulation, coupling the abilities offered by the original systems with the convenience of integration with sequencer packages and without the need for new hardware.

The applications that will now be created for Chapter 2 will demonstrate many of the benefits of core advances discussed in this chapter.

7 Chapter 2 - Use the Steinberg SDK to create an advanced ring modulating plugin for use in host-based systems

In this chapter the development cycle of an advanced ring modulating plugin for the VST host system is described. This development cycle is divided up into 4 sub-stages, with each stage building upon the ideas presented in the last.

The sub-stages are as follows:

1- A Plugin that has no effect

2- A Sawtooth ring modulator

3- A Sine-Wave Ring Modulator

4- A Sine-Wave Ring Modulator with additional LFO control

Although Steinberg's SDK takes control of much of the interaction between the plugin and the host, all algorithms presented in this section are completely original works designed specifically for this project report.

SDK availability

Steinberg, a software company based in Germany, are the manufacturers of Cubase VST - one of the foremost host-based sequencers available. Since their software sales benefit from the availability of plugins, they encourage 3rd parties to develop them. In order for all plugins to work correctly with Steinberg's host, an SDK is available that gives instructions on where to begin and how to maintain compatibility.

This SDK should be acquired as a first step towards creating a new plugin. At time of writing, the latest version (2.0) of the SDK is available at http://www.steinberg.net/.

Plugin Foundations

7.1.1 Creating a *.dll

A VST plugin is a standalone *.dll file which is placed in a directory to which the VST host is directed. It was found that details of how this works vary from host to host, so for details of this it is best to check the host's manual.

The fastest way to begin creating new plugins is to open the default workspace provided by the SDK, "vst2examples.dsw". For the purposes of this report example plugins were taken as starting points in order to achieve the correct infrastructure. The provided algorithms and GUI control settings where deleted and new controls added as needed.

The plugin is written using the C++ computer language. A good understanding of C++, object oriented programming, and DSP techniques was found to be indispensable since these skills are all needed to conceive and implement new plugin ideas.

7.1.2 Inheriting Elements of the SDK

The SDK comes with many *.h, *.hpp, and *.cpp files already placed in the default workspace. Of all these files, it is only one *.cpp file that will be focused upon. Steinberg demand that the other included files are inherited not edited.

"Never edit any of these files. Never ever. The host application relies on them being used as they are provided." (Steinberg, 2001 [9])

7.1.3 GUI

Although a custom GUI can be created for 3rd party plugins, in the absence of such a GUI, the host will automatically generate a default visualisation for the controls. For the purposes of this project, the default host GUI was used since the focus of the project is on the audio signal processing rather than graphical design. A second benefit of accepting the default GUI is that the plugin will remain platform independent.

"(The SDK) allows practically any user interface to be defined. A negative aspect is that then you can quickly land up in platform specifics when dealing with the nuts and bolts of the interface issues." (Steinberg, 2001. [9])

The technique of visualisation varies from host to host. For example Steinberg's Cubase VST represents user variables as horizontal sliders, Sonic Syndicate's Orion Pro represents them as knobs, and Steinberg's WaveLab represents them by simulating a rack-mounted device. Below are screenshots of the completed advanced ring modulator plugin as seen in these three different hosts:

Figure 7-1 Sonic Syndicate's Orion default plugin GUI

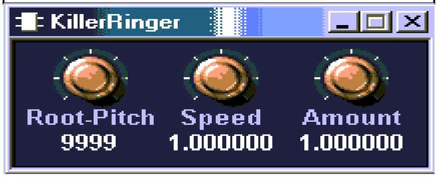


Figure 7-2 Steinberg's WaveLab default GUI

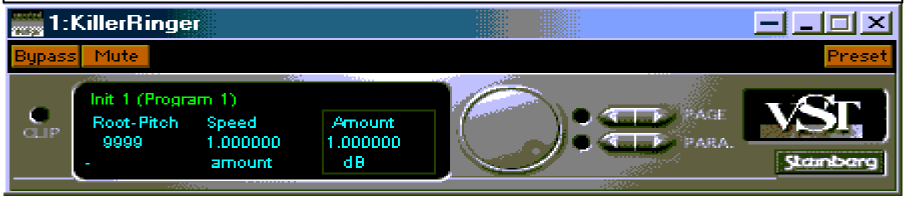
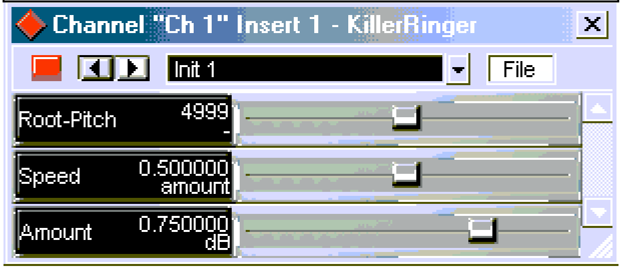


Figure 7-3 Steinberg's Cubase default GUI


Channel "Ch 1" Insert 1 - KillerRinger

Init 1

File

Root-Pitch	4999	
Speed	0.500000	amount
Amount	0.750000	dB

7.2 <u>Development of proposed applications</u>

7.2.1 <u>Create a plugin that has no effect.</u>

It is unwise to begin implementing complicated algorithms from the offset. For this project it was found that to get to grips with the SDK it was better to start with something very simple and ensure that it was understood completely. From this foundation, more complicated effects can then be created.

The most basic plugin is a plugin that does nothing, i.e. has no effect upon the inputted sound.

This example of a plugin with no effect will help to describe some of the functions involved. An understanding of these will be needed when creating more complicated applications such as those found later in the report.

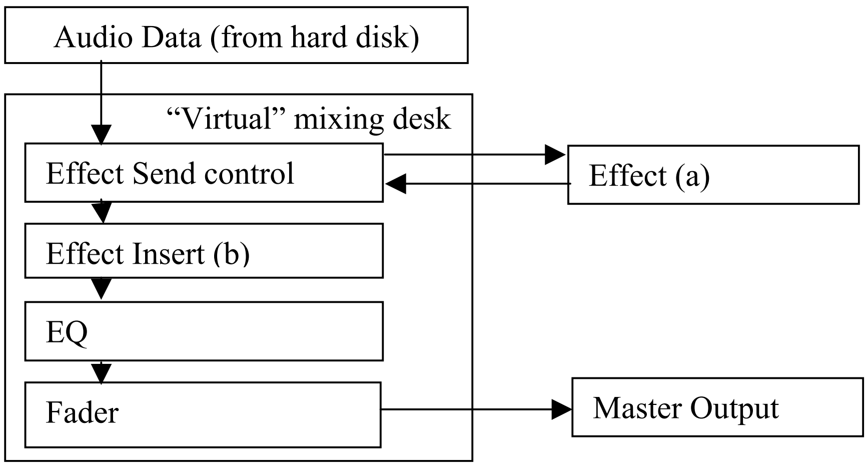
7.2.1.1 The process() and processReplacing() functions

The host application that plays back audio files is often modelled upon the classic hardware mixer interface design. The audio is streamed from the hard disk through a virtual mixer channel, and that channel includes associated "send" and "insert" settings in similar vein to hardware mixers. It is at these two points that plugin effects can be added. The "send" effect allows a proportion of the inputted sound to be conveyed to the plugin while the unaffected audio continues to flow straight to the master output. This allows the amount of effect used (i.e. the wet/dry mix) to be controlled by the "send" control.

In contrast, the "insert" effect sends the entire audio signal through the plugin, removing any control of wet/dry balance.

This is explained diagrammatically overleaf:

Figure 7-4 Simplified model of a virtual mixing desk signal path



The choice of "send" or "insert" operation within the plugin code is accomplished using the process() and processReplacing() functions held in the audioeffect and audioeffectx header files. These functions act as intermediaries between the plugin and the host, and actually represent the "send" and "insert" effect-types respectively.

For this first example, the following code was constructed. The plugin is designed to take audio from the input and convey it directly to the output. As such, it operates as an insert effect, and should be placed within the processReplacing() function.

(The *.cpp file in which this function is placed is somewhat lengthy, so has been included in the appendix for reference.)

Figure 7-5 processReplacing source code for a plugin with no effect

```
void  AGain::processReplacing(float  **inputs,  float
**outputs, long
 sampleFrames)
 {
     float *in1  =  inputs[0];
     float *in2  =  inputs[1];
     float *out1 = outputs[0];
     float *out2 = outputs[1];
        while(--sampleFrames >= 0)
            {
              (*out1++) = (*in1++) ;
              (*out2++) = (*in2++) ;
            }
   }
```

7.2.1.2 Discussion of processReplacing() Source code for a plugin with no effect

The elements of this function are as follows:

When the function is begun, it is fed **inputs and **outputs, which direct it to the input buffer and the output buffer. SampleFrames denotes the size of the buffer.

The input and output values are then made equal to variables in1 and in2 (being the left and right channels of the input) and out1 and out2, (left and right channels of the output)

The while loop that follows decrements sampleFrames until it is zero, each time copying a value from the input to the output, and moving both input and output along one sample (by using the increment function "++")

Once sampleFrames = 0, the buffer has been completed, and the function finishes. This entire process is then repeated with a new buffer-load of audio.

This basic operation is not discussed in the SDK documentation and it was necessary to study C++ reference books in order to gain understanding of the pointers, arrays and buffers involved.

7.2.2 Audio representation within the SDK

Since this detail is not well documented in the Steinberg SDK, the following experiment was devised and carried out to discover that plugins represent audio as values between +/-1.

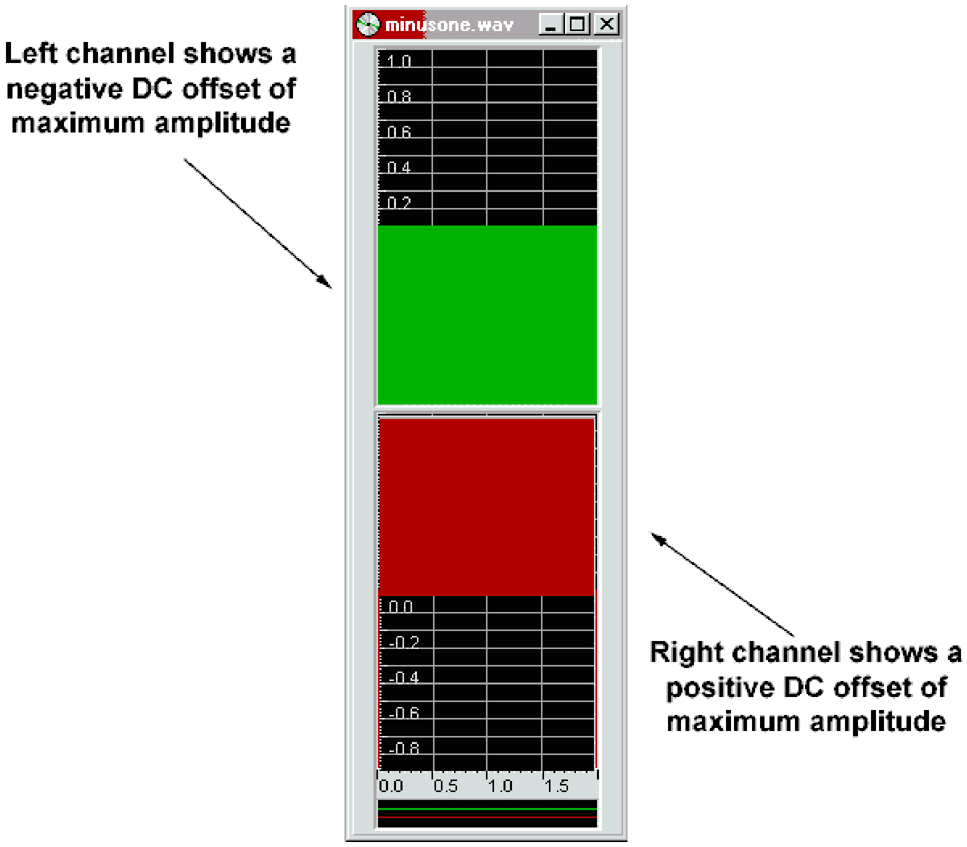
Firstly, a plugin that outputs a constant +1 to one stereo channel, and –1 to the other, was created. The code written to achieve this is shown overleaf.

Figure 7-6 Algorithm code used to test audio representation

```
void AGain::processReplacing(float **inputs, float **outputs, long
 sampleFrames)
 {
     float *in1  =  inputs[0];
     float *in2  =  inputs[1];
     float *out1 = outputs[0];
     float *out2 = outputs[1];
        while(--sampleFrames >= 0)
            {
              (*out1++) = (1) ;
              (*out2++) = (-1) ;
            }
   }
```

Compiling this code and recording the audio that is outputted by the host when the plugin is in operation creates the following results:

Figure 7-7 Audio output from previous figure as displayed in a wave editor


Left channel shows a negative DC offset of maximum amplitude
minusone.wav
1.0
0.8
0.6
0.4
0.2
0.0
-0.2
-0.4
-0.6
-0.8
0.0
0.5
1.0
1.5
Right channel shows a positive DC offset of maximum amplitude

This demonstrates that audio is represented between +/-1. This information will be invaluable when deriving algorithms later.

7.2.3 <u>Create a sawtooth ring modulator</u>

As the next step on from the "blank" plugin, detailed earlier, a sawtooth ring modulator will now be constructed.

This example will give a foundation understanding of a plugin that can be augmented with new features as desired.

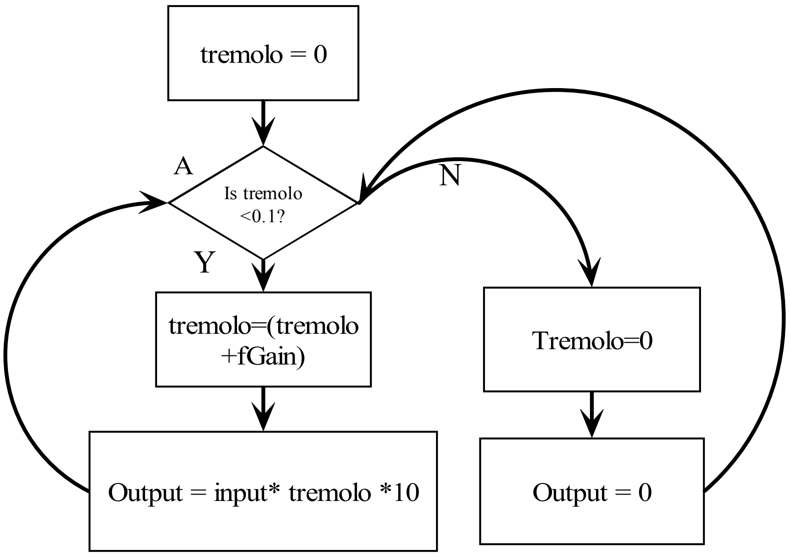
Software planning – Operation flow charts of the sawtooth ring modulator

When actually creating the DSP algorithm in code, it is very important to follow strict software engineering techniques in order to remain in full understanding of the code that has been written.

Before typing a single line of code, a system design must be built. "The aim of system design is to develop a gross architecture of a system" (Ince, 1994 [2])

The design that was conceived for this first algorithm is shown here.

Figure 7-8 A flowchart describing a sawtooth ring modulator algorithm



It was decided that the best way to create a sawtooth wave was to use a counter that resets upon reaching a defined value.

Firstly, a variable "tremolo" is declared and set to zero. It is then checked to see if it is greater than 0.1. Zero isn't, so "tremolo" is incremented by amount: "fGain", a variable that is adjusted by the user.

fGain can be linked to a slider or knob on the GUI.

The inputted audio sample is multiplied by tremolo.

This stage causes a loss of gain by a factor of 10, so it was decided to make this gain up by multiplying the result by 10. This final result is then sent to the output.

The program returns back to “A”, and continues as described above until tremolo becomes >=0.1. At this time, tremolo is re-set to zero, and the plugin outputs a sample of zero magnitude. From here, the program once again returns to point “A” and the entire process repeats.

The predicted output of this plugin is shown here modelled in Microsoft Excel.

Figure 7-9 Sine wave input to the plugin

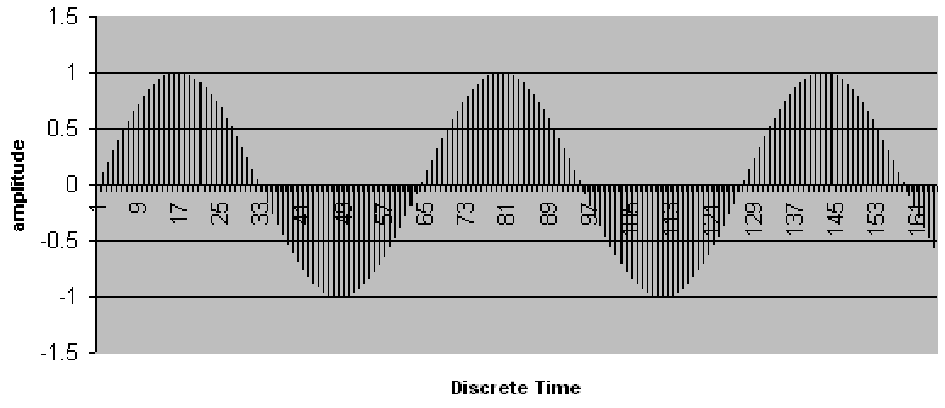


Figure 7-10 Sawtooth modulation signal

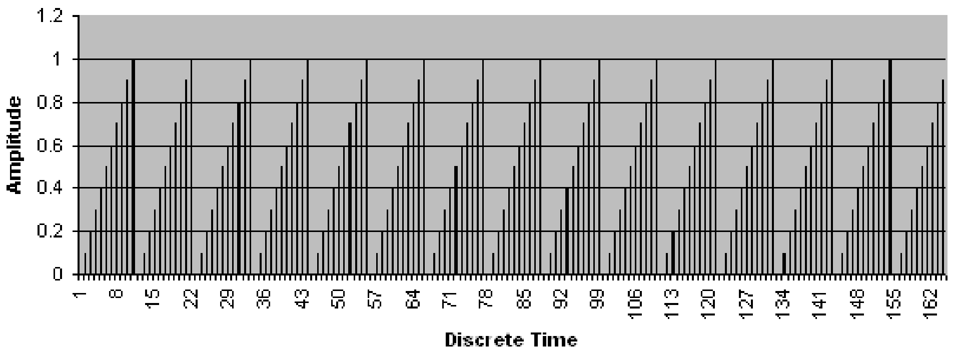
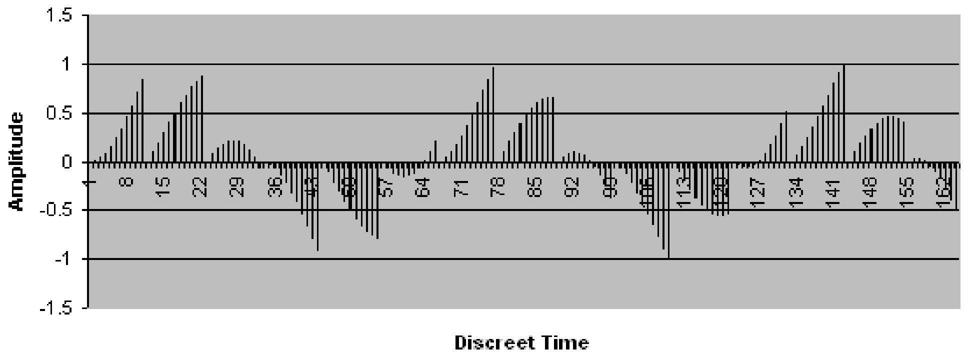


Figure 7-11 Anticipated output of plugin



The realisation of this software design was created in code as follows: (only the algorithm is shown here, the full source is available in the appendix)

Figure 7-12 processReplacing source code for a sawtooth ring modulator plugin operating at arbitrary frequencies

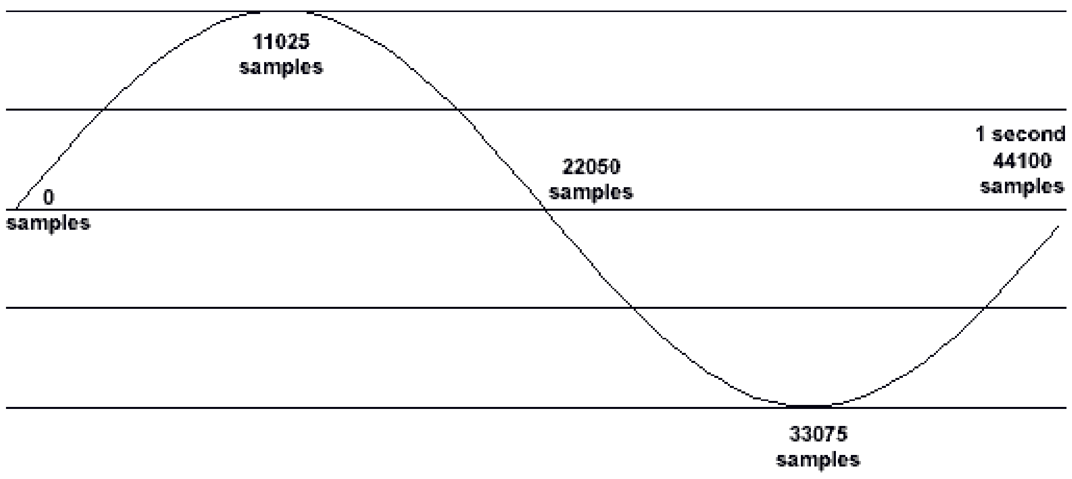
```
double tremolo=0;
void AGain::processReplacing(float **inputs, float **outputs, long
 sampleFrames)
 {
    float *in1  =  inputs[0];
    float *in2  =  inputs[1];
    float *out1 = outputs[0];
    float *out2 = outputs[1];
  if ( tremolo < 0.1 )
      {
      tremolo=(tremolo+fGain);
      while(--sampleFrames >= 0)
          {
           (*out1++) = (*in1++) * tremolo *10;
           (*out2++) = (*in2++) * tremolo *10;
          }
      }
  if ( tremolo >= 0.1 )
   {
    tremolo=0;
      while(--sampleFrames >= 0)
      {
           (*out1++) = (*in1++) * tremolo;
           (*out2++) = (*in2++) * tremolo;
      }
   }
}
```

This code will modulate the input using a sawtooth wave at an arbitrary frequency set by the user. The variable “tremolo” is declared outside of the processReplacing() function to avoid being re-initialised each time a buffer of audio is completed. The other elements of the code are as per the software design flowchart Figure 7.8: “A flowchart describing a sawtooth ring modulator algorithm”.

To allow the user to modulate at a specific chosen frequency rather than an arbitrary frequency, some additions must be made to the code. Since the host software works at 44100Hz, (i.e. 44100 samples per second) this value can be used to derive the length of one Hz in C++ code.

The sketch below was created to show one period of a sine wave, at 1Hz, divided up into 44100 samples:

Figure 7-13 Sketch to illustrate the division of a 1Hz wave period into individual samples



$$\frac{44100}{1} = 44100 = 1 \text{ second}$$

$$\frac{44122}{2} = 22050 = 1/2 \text{ second}$$

$$\frac{44100}{4} = 11025 = 1/4 \text{ second}$$

$$3\text{x} \; \frac{44100}{4} = 3075 = 3/4 \text{ second}$$

Deriving the above values indicated how many digital samples are processed each ½, ¼, ¾, and full second. This supplies the relevant information on how to derive specific frequencies in C++. If the tremolo variable used in the sawtooth code (re-named to "counter" herein for clarification) is adjusted such that it is increased by a 11020th of 1, for each sample that is processed, it will reach a fixed proportion of 1 every 8th of a second. ("Counter" grows twice as quickly when processing stereo wave files due to an increase of "counter" once per sample for each channel.)

1/11020 = 0.000090744

If this value (11020th of 1) is then divided by 8, "counter" will increase and reset at the correct speed to generate a 1Hz sawtooth wave.

0.000090744 / 8 = 0.000011343 increase needed per sample at "counter"

This value can then used to give the user control over the modulation frequency. A new variable "topFreq" is introduced that denotes the maximum frequency of the modulator. (It is set at 1kHz since with experimentation it was found that this offers the most creatively useful range of modulation.) The user variable "fGain" that creates values between 0 and 1 can be multiplied by the topFreq variable to generate proportions of 1kHz on the fly.

The algorithm code that contains these modifications is shown below:

(The complete *.cpp file can be found in appendix)

Figure 7-14 Algorithm code allowing user to select specific modulation frequencies.

```
float counter = 0;
double quarterSec = 0.000011343;
double topFreq = 1000;
void AGain::processReplacing(float **inputs, float **outputs, long
 sampleFrames)
{
    float *in1  =  inputs[0];
    float *in2  =  inputs[1];
    float *out1 = outputs[0];
    float *out2 = outputs[1];
        while(--sampleFrames >= 0)
            {
                    counter = (counter + (quarterSec*(topFreq*fGain)));
                    if (counter > 0.5)
                           {
                           counter = 0;
                           }
              (*out1++) = (*in1++ * counter);
              (*out2++) = (*in2++ * counter);


            }
}
```

7.2.4 <u>Replacement of the sawtooth modulator with a sine wave</u>

A sawtooth modulator is not ideal due to the rich harmonic content caused by steep transients in sawtooth waves. These harmonics, when ring-modulated with a second signal, produce double the number of harmonics again due to the "sum and difference" nature of ring modulation. The abundance of new harmonics can overpower any pitch information in the source signal.

The use of a sine wave as a modulator will produce fewer harmonics and a more musical sound.

A sine wave can be generated in our algorithm by utilising the "counter" variable. By adjusting the code used previously such that "counter" counts from 0 to 2pi, a sine wave can be generated from the resultant series of numbers.

It was found that "counter" must count <u>precisely</u> from 0 to 2pi in order to create a perfect single period of sine before it is reset. If "counter" over-counts by the slightest amount, distortion of the modulation signal will occur.

The following diagrams show the adverse effects that occurred as a result of over-counting. The first diagram shows a portion of the sine wave with the extra sample identified. The second diagram shows an FFT of a pure sine wave followed by the distorted sine wave.

Figure 7-16 Detail of a 1/2 period sine wave showing 1 errant sample

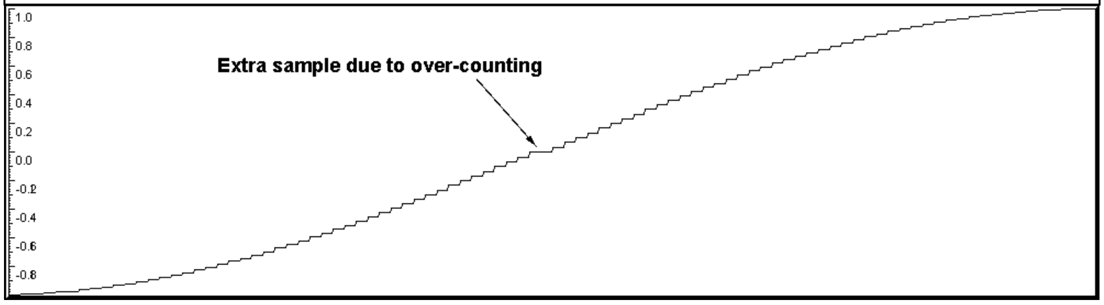
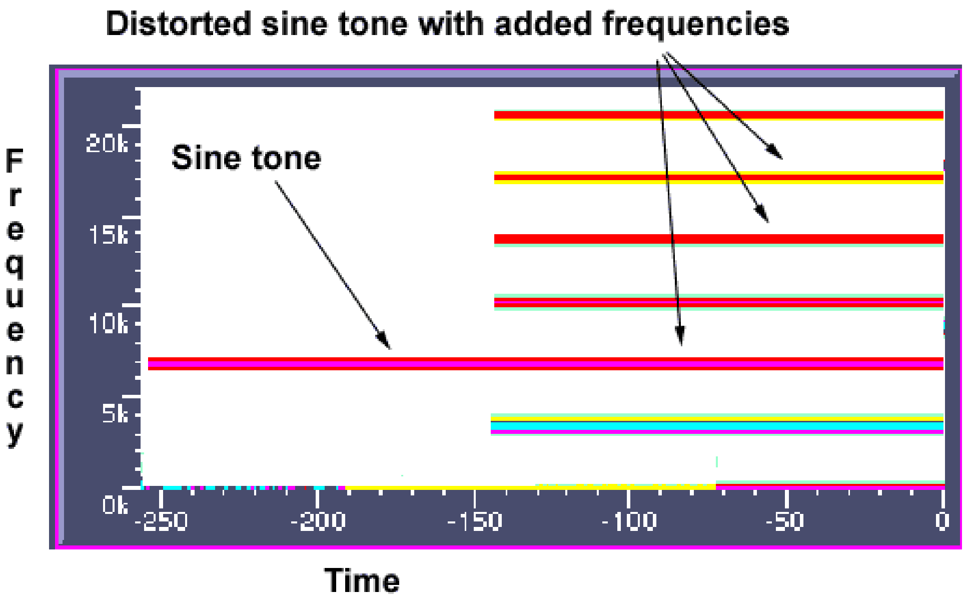


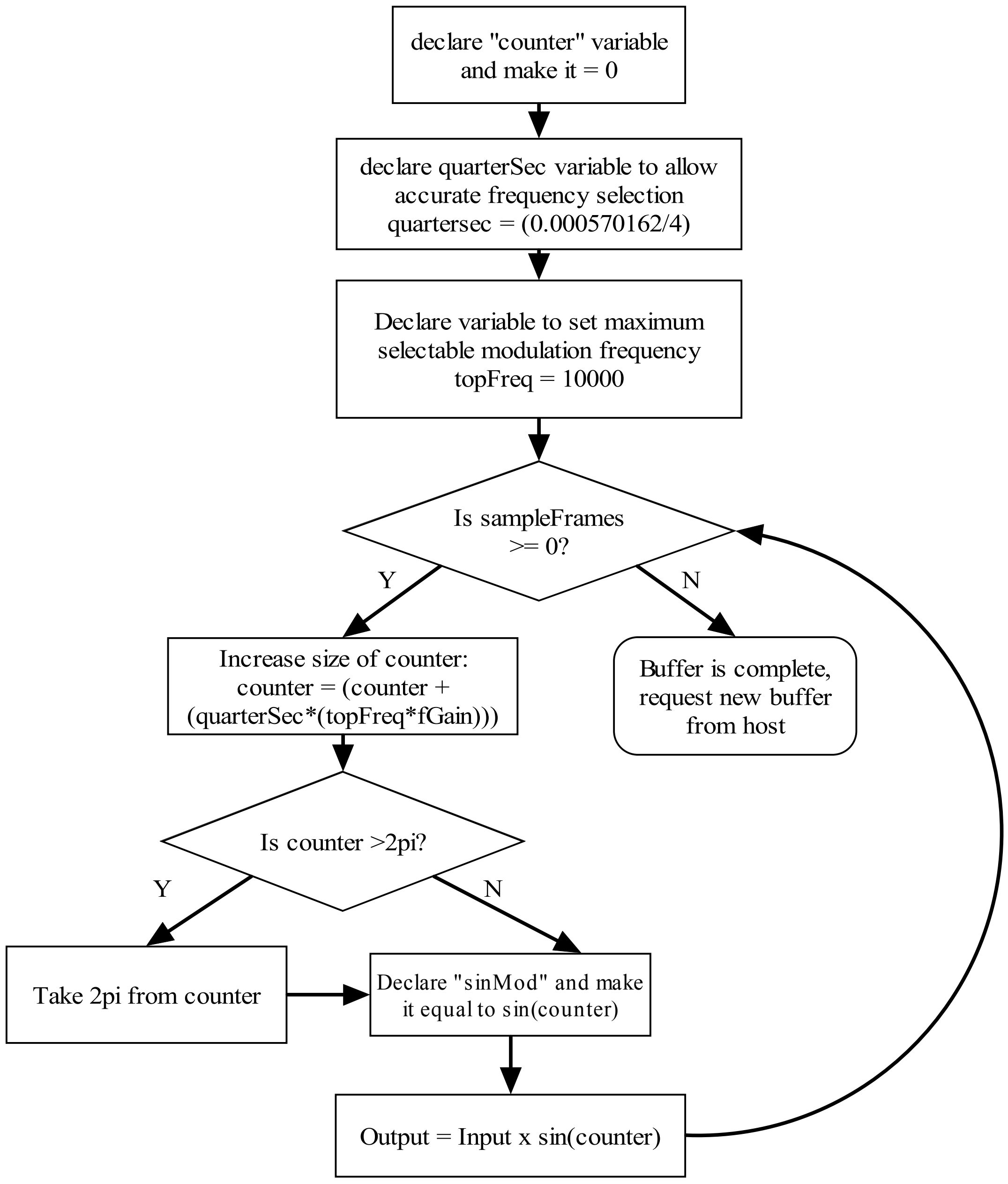
Figure 7-15 FFT comparing a pure sine tone with the wave displayed above



By conducting this experiment it is shown that by just adding a single errant sample to the sine signal, the FFT contains many new harmonics where there should be only one fundamental frequency present.

With these points in mind the following flowchart for the sine wave ring modulator algorithm was designed.

Figure 7-17 Flowchart detailing operation of a sine wave ring modulator algorithm



declare "counter" variable and make it = 0
declare quarterSec variable to allow accurate frequency selection quartersec = (0.000570162/4)
Declare variable to set maximum selectable modulation frequency topFreq = 10000
Is sampleFrames >= 0?
Y
N
Increase size of counter: counter = (counter + (quarterSec*(topFreq*fGain)))
Buffer is complete, request new buffer from host
Is counter >2pi?
Y
N
Take 2pi from counter
Declare "sinMod" and make it equal to sin(counter)
Output = Input x sin(counter)

This flow chart is more complex than previous examples. It shows the operation of the sampleFrames "while loop", a loop that controls the use of audio buffers from the host. The variable sampleFrames holds the number of samples in each buffer-load of audio. For each sample that is processed, sampleFrames is decremented and checked to see if it has reached zero. Upon reaching zero, the buffer is empty and a new full buffer is requested from the host.

The predicted output of this sine ring modulator plugin is shown here as modelled in Microsoft Excel.

Figure 7-18 A sine wave input to the plugin

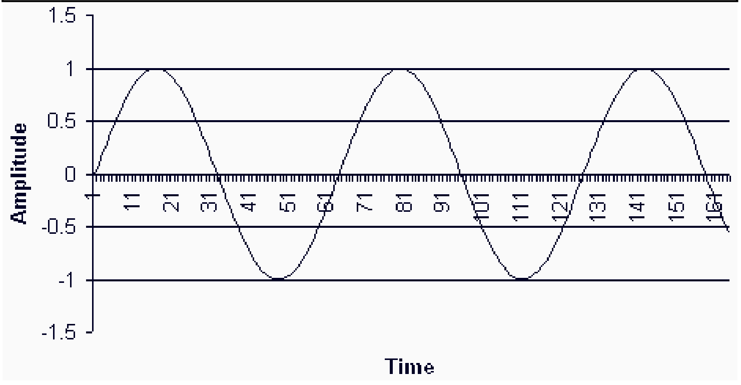


Figure 7-19 A high frequency sine wave used to modulate the input wave

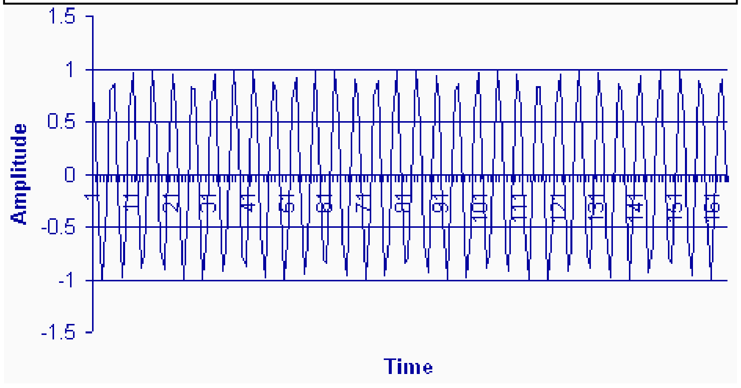
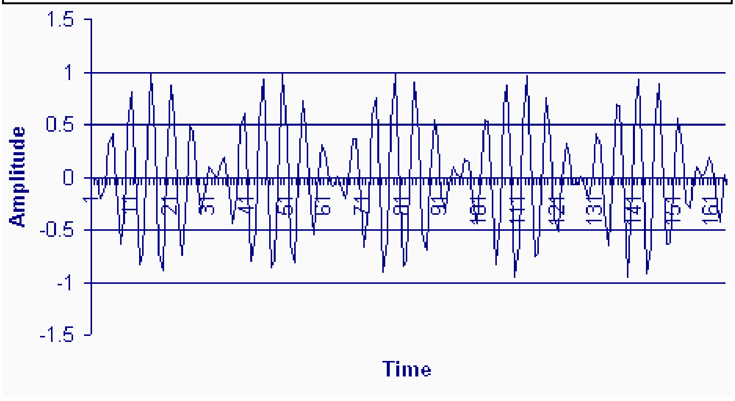


Figure 7-20 The resultant output of the plugin



The realisation of this software plan is shown in the segment of code below. (The complete sine wave ring modulator code can be found in the appendix)

Figure 7-21 Algorithm code for a sine wave ring modulator

```
float counter = 0;
double quarterSec = (0.000570162/4);
double topFreq = 10000;
void  AGain::processReplacing(float  **inputs,  float
**outputs, long
 sampleFrames)
{
   float *in1  =  inputs[0];
   float *in2  =  inputs[1];
   float *out1 = outputs[0];
   float *out2 = outputs[1];
     while(--sampleFrames >= 0)
     {
     counter = (counter+(quarterSec*(topFreq*fGain)));
     if (counter > (2*3.14159))
          {
          counter -= (2*3.14159);
          }
     double sinMod = sin(counter);
     (*out1++) = ((*in1++)*(sinMod/2));
     (*out2++) = ((*in2++)*(sinMod/2));
     }
}
```

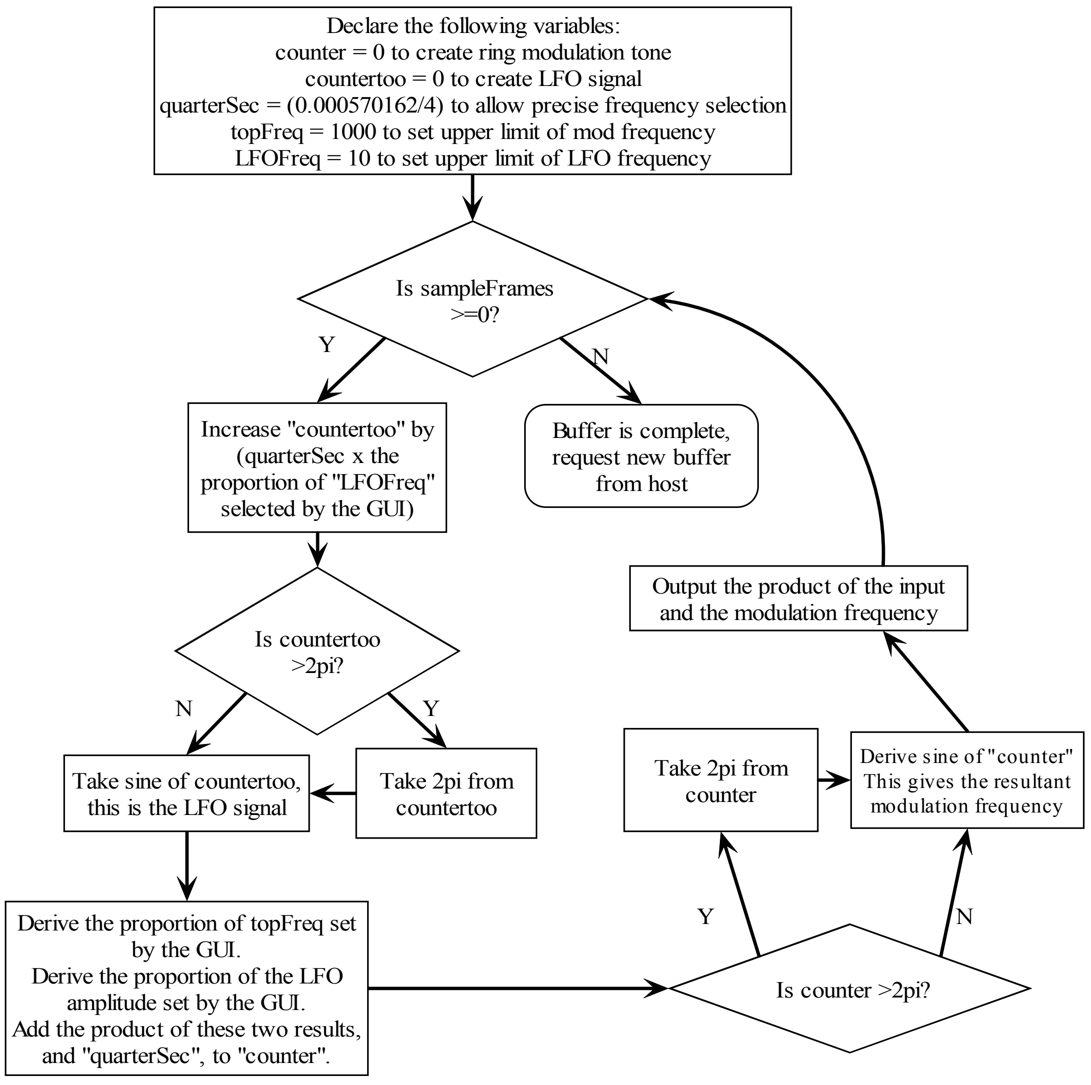
7.2.5 Add further control for the user

As a final addition to the plugin, an LFO will be introduced to control the ring modulation frequency. Two dials will be added to the GUI to control LFO frequency and LFO depth.

This control will improve the plugin's repertoire of sounds and as such it's creative applications.

The software flowchart plan designed to implement this improvement is shown overleaf:

Figure 7-22 Software Design for an LFO controlled Ring Modulator


Declare the following variables:
counter = 0 to create ring modulation tone
countertoo = 0 to create LFO signal
quarterSec = (0.000570162/4) to allow precise frequency selection
topFreq = 1000 to set upper limit of mod frequency
LFOFreq = 10 to set upper limit of LFO frequency
Is sampleFrames >=0?
Y
N
Increase "countertoo" by (quarterSec x the proportion of "LFOFreq" selected by the GUI)
Buffer is complete, request new buffer from host
Is countertoo >2pi?
N
Y
Take sine of countertoo, this is the LFO signal
Take 2pi from countertoo
Output the product of the input and the modulation frequency
Take 2pi from counter
Derive sine of "counter" This gives the resultant modulation frequency
Derive the proportion of topFreq set by the GUI. Derive the proportion of the LFO amplitude set by the GUI. Add the product of these two results, and "quarterSec", to "counter".
Y
N
Is counter >2pi?

The diagram shows the generation of two separate sine signals. One is the modulation frequency and is created in a similar way to earlier examples.

The second is the LFO generator. The user will have control over the amplitude and the frequency of this signal, by linking the two new user-controlled variables to the increase-rate of "countertoo".

From this software plan the following code was written. Only the algorithm is shown here, the complete source is found in the appendix.

Figure 7-23 Algorithm Source code written to implement an LFO controlled Ring Modulator

```
float counter = 0;
float countertoo = 0;
double quarterSec = (0.000570162/4);
double topFreq = 10000;
double LFOFreq = 10;
void ADelay::processReplacing(float **inputs, float **outputs, long
 sampleFrames)
{
   float *in1  =  inputs[0];
   float *in2  =  inputs[1];
   float *out1 = outputs[0];
   float *out2 = outputs[1];
      while(--sampleFrames >= 0)
      {
         countertoo = (countertoo + (quarterSec*(LFOFreq*fFeedBack)));
         if (countertoo > (2*3.14159))
                  {
                  countertoo -= (2*3.14159);
                  }
         double nuMod = sin(countertoo);
         counter = (counter + (quarterSec*(topFreq*fDelay)+(nuMod*(fOut/8))));
         if (counter > (2*3.14159))
                  {
                  counter -= (2*3.14159);
                  }
         double sinMod = sin(counter);
         (*out1++) = ((*in1++)*(sinMod/2));
         (*out2++) = ((*in2++)*(sinMod/2));
         }
}
```

7.2.6 <u>Details of miscellaneous techniques and additions to code</u>

The focus of this chapter is on the planning and development of new algorithms. It is not intended to comprehensively cover every line of code, modification, or step involved in implementing chosen algorithm due to space constrictions. There are, however, some more important core changes that must be made to the SDK that will be mentioned in this section:

7.2.6.1 Enumerating new GUI controls

Study of C++ enumeration techniques was carried out in order to introduce extra variables to the GUI. It was understood that the enumerated type allows a set of possible values to each be given a descriptive name allowing the clear differentiation between them. Enumeration is ideal for declaration of new GUI variables, and is done by using the phrase:

Figure 7-24 Enumeration Technique

```
enum
{
        variable1;
        variable2;
        variable3;
        kNumParams
};
```

This technique allows the host to have clear identification of the dials and sliders it needs to generate at runtime, and also makes source code more manageable due to the identification of confusing numeric variables with “friendly” variable names.

7.2.6.2 Labelling variables in the GUI

Many of the more important miscellaneous lines of code needed to create new plugins rely upon calling functions from the SDK. For example, in order to mark the GUI representation of user-alterable variable controls with meaningful labels, a function “getParameterName” is called from audioEffect.CPP

This function is sent two values; an index denoting which variable is to be labelled as per the enumeration index, and a pointer to a char data location in which the parameter name is to be stored.

Figure 7-25 Usage of the "getParameterName" function

```
void ADelay::getParameterName (long index, char *label)
{
        switch (index)
        {
                case kDelay :    strcpy (label, " Root-Pitch  "); break;
                case kFeedBack : strcpy (label, "Speed"); break;
                case kOut :      strcpy (label, " Amount "); break;
        }
}
```

Above, three dials enumerated as “kDelay”, “kFeedBack” and “kOut” are labelled with the strings “Root-Pitch”, “Speed” and “Amount” respectively.

The labels that result from this code can be seen in Fig. 7.1 “Sonic Syndicates’s Orion default plugin GUI” earlier in this chapter.

7.2.6.3 Unique names

Plugins must have unique names in order to allow the host to differentiate between them. A function called setUniqueID() is used to give the plugin it’s unique name, and upon choosing a name for a plugin it is possible to contact Steinberg and check that the name has not been used before. The name chosen for this project was “Kllr”, which is set by adding the following line at plugin initialisation:

```
setUniqueID ('Kllr');
```

8 Analysis and Discussion

8.1 Spectrograph Testing

The main aim of this project is to create an advanced ring modulation plugin for the VST system. Tests will now be carried out in order to verify that the code completed in the previous section is operating correctly as such.

The nature of a ring modulator is that if it is fed with one input frequency, it will output two frequencies. One will be the sum of the input frequency and the modulation frequency, and the other will be the difference between the input frequency and the modulation frequency.

I.e. if it is fed with a 5kHz test tone and the modulation frequency is set to 2.5kHz, then the following two signals will be outputted:

5kHz + 2.5kHz = **7.5kHz**

5kHz – 2.5kHz = **2.5kHz**

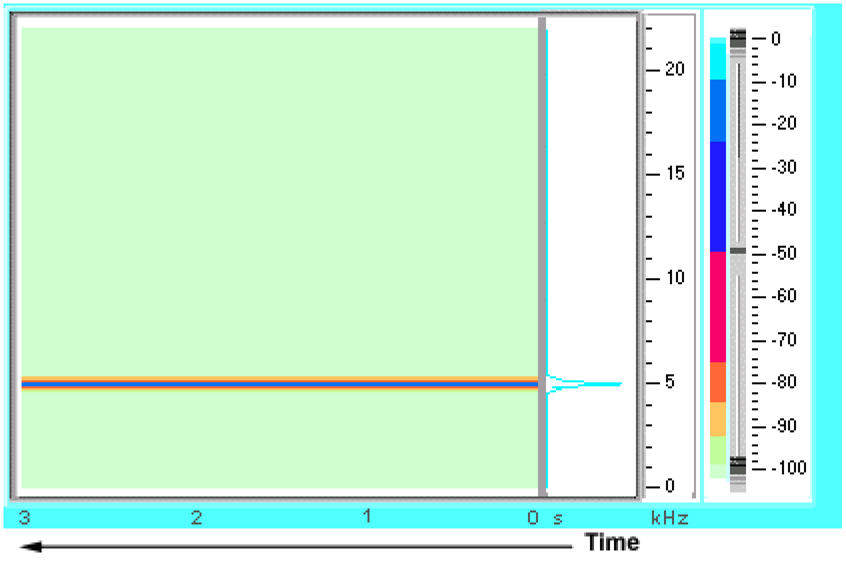
By the same methods it is derived that if the input is fed with a 5kHz test tone and the modulation frequency is set to 5kHz, then the output will consist of a 10kHz tone plus a 0Hz DC offset.

In order to test the plugin, a signal generator was used to create a 5kHz test tone. The plugin was compiled and loaded into Steinberg's WaveLab as its host.

By playing the generated 5kHz tone through the ring modulating plugin, it was possible to analyse the output of the plugin using a spectrograph.

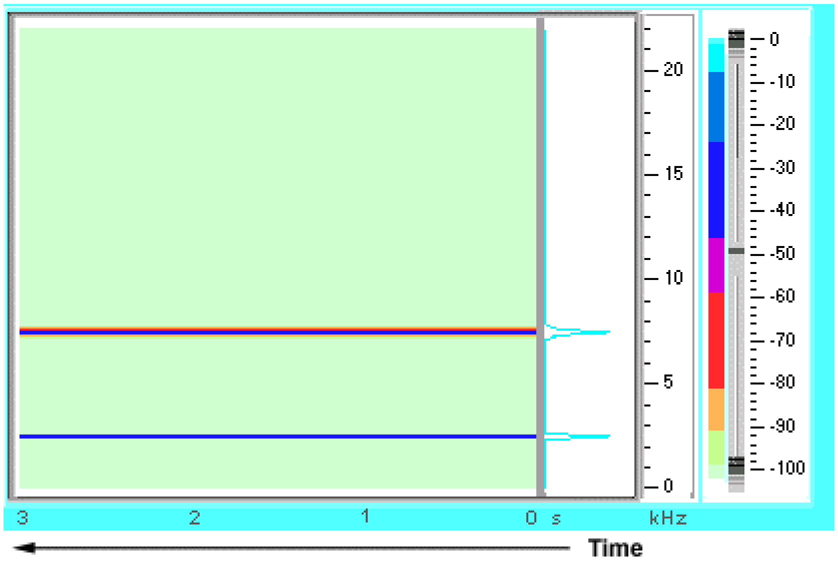
Firstly a spectrograph was taken of the input signal, to ensure that it was 5kHz. This is shown below.

Figure 8-1 Spectrograph of the 5kHz test tone



Then, the plugin was set to modulate at 2.5kHz and the output was analysed:

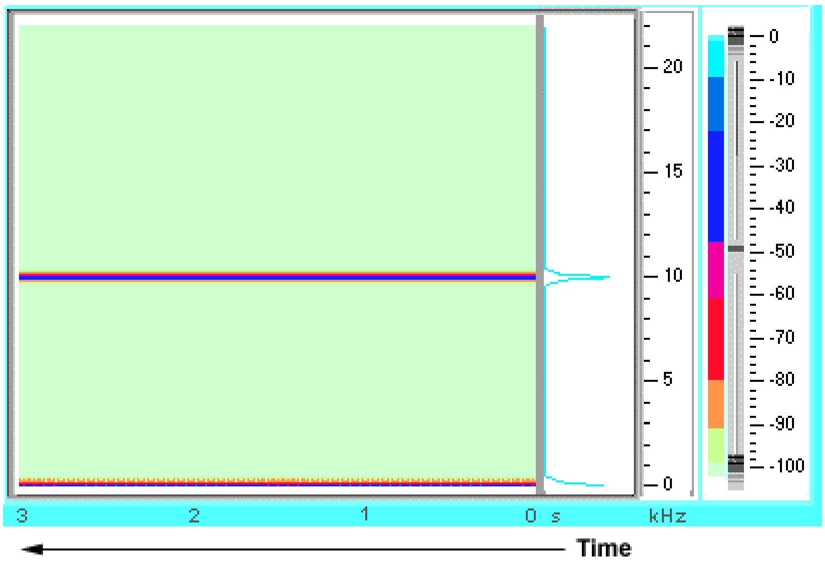
Figure 8-2 Spectrograph of a 5kHz tone ring modulated by a 2.5kHz tone



From this spectrograph it can be seen that the output consists of a 7.5kHz and a 2.5kHz tone.

To verify these results, a second test was carried out with a 5kHz test tone and an identical 5kHz modulation frequency. A spectrograph of the output is shown here:

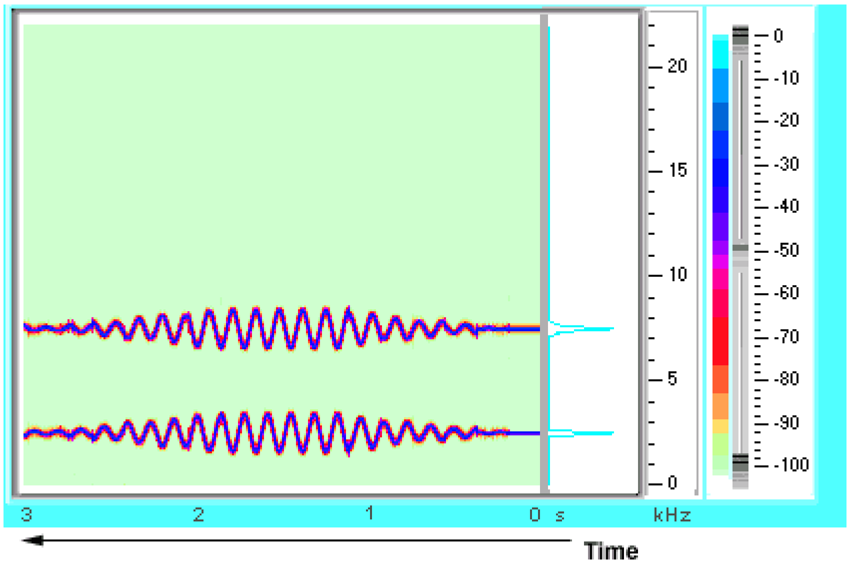
Figure 8-3 Spectrograph of a 5kHz tone ring modulated by a 5kHz tone



As expected, the output consists of a 10kHz tone and a 0Hz DC offset. These results conclusively show that the ring modulation algorithm created for this report works correctly.

To verify that the LFO code that controls the modulation frequency is correct, the same 5kHz tone was fed through the ring modulator with a 2.5kHz modulation frequency. While the tone played, the LFO depth control was increased from zero to maximum and then returned to zero. This spectrogram shows the results:

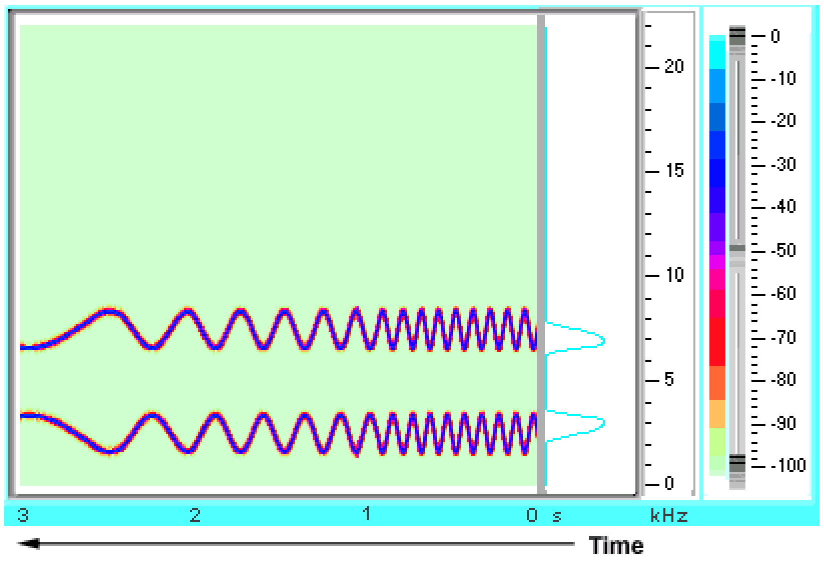
Figure 8-4 Spectrograph showing adjustment of the LFO depth



As anticipated, the output frequencies start identically to those in Figure 8.2 “Spectrograph of a 5kHz tone ring modulated by a 2.5kHz tone”, but begin to oscillate as the LFO depth control is increased. As the LFO depth decreases, the oscillations reduce and the output becomes once again constant.

The final test was concerning the LFO frequency control. The same 5kHz tone was fed through the ring modulator with a 2.5kHz modulation frequency and LFO depth set to maximum. While the tone played, the LFO frequency control was increased from zero to maximum. The spectrogram overleaf shows the results:

Figure 8-5 Spectrograph showing changes in LFO frequency



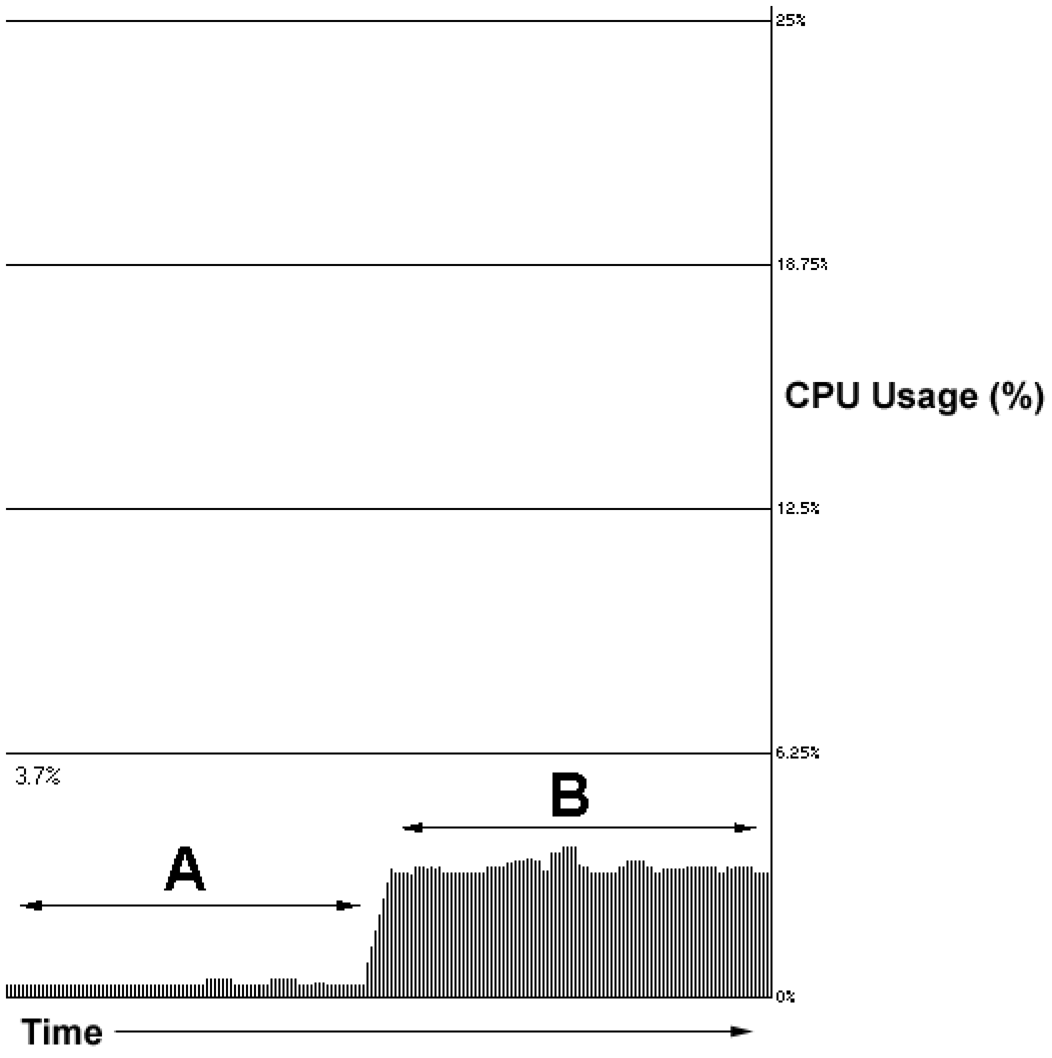
Once again as anticipated, the output frequency oscillation begins as a constant, but as time goes on it oscillates faster and faster, with no change in root modulation frequency or LFO depth.

Of particular interest is the purity of the tones generated from the ring modulator. There are no erroneous aliases and there is no noise visible on the spectrograph. This shows that the mathematics used in the C++ design of this application were accurate down to sample level.

8.2 <u>Efficiency of code</u>

The efficiency of the code is very important due to the system intensive nature of real time audio playback. A plugin that takes up too many system resources can deny other parts of the system the CPU cycles they need to produce smooth glitch-free sound. Plugins available at time of writing often use between 1% and 15% of the available CPU cycles to carry out their effects when used on a 550MHz Pentium III. The ring modulator designed for this project was tested for its efficiency by running it from Steinberg's WaveLab and monitoring the CPU usage as the plugin was enabled and disabled. The following data was recorded:

Figure 8-6 Graph showing CPU usage against time



This graph shows CPU usage while the plugin is disabled (section A) and enabled (section B). When disabled, CPU usage is approximately 0.5%, and when enabled it is approximately 3.7%. Therefore the ring modulator plugin uses approximately 3.2% of the CPU cycles on a 550MHz Pentium III system. This is in no way too inefficient. However, if it had been necessary to improve efficiency, this could have been done by the use of sine wave look-up tables. These store the sampled values of a sine wave and negate the need to perform sine calculations on the fly, thus reducing processing time of each sample.

8.3 Usability Research – Comments

Correctly functioning code and good efficiency mean nothing if the plugin creates an unusable non-musical sound. In order to get a measure of the plugins usability, a website where the plugin could be downloaded was created for the purposes of this project, alongside a request for feedback. The website content is included in the appendix.

At time of writing, the plugin created for this project has been downloaded 139 times (Beseen Hit counter statistics, 2002 [l]). All feedback received thus far is positive, and has been included in the appendix. Highlights are shown here:

“It's neatly done, all the parameters have nice control feel with no crackling or stuff.”
(Vesa Norila)

“very artistic sound :o)”
(Igor Yael)

“i have loaded it up in orion, ooh it sounds all wibbly and metallic – wibblytastic!”
(Paul Stimpson)

“yup, d/l'd your plug-in and it works well in audiomulch (a nice effect- given the title i was half expecting a run of the mill ring modulator ;-)”
(Paul Random99)

This feedback indicates that the plugin design has been very successful.

9 Implications of Change

Backward compatibility

When the VST 1.0 system was upgraded to VST2.0, the original files were expanded upon rather than replaced. As the VST system improves over time, Steinberg state in their SDK documentation that will try to maintain this backward compatibility.

Software synthesis begins to replace it's hardware counterpart

Since so much of computer based audio processing is currently achieved by use of software instead of hardware, the hardware synthesis features of soundcards are becoming less important. The focus of soundcards seems now to be more on the quality of reproduction. Recently released soundcards such as the MidiMan AudioPhile 24/96, that uses a 24-bit 96 kHz DAC, focus on greater bit depth, higher sample frequency support, and digital I/O, rather than the new synthesis options that were the focus of earlier soundcards such as the Yamaha Adlib.

10 Conclusions

How does the practical element in Chapter 2 successfully demonstrate the core benefits of many of the new technologies discussed in the Chapter 1?

The analysis and discussion section of this report demonstrates that the first aim, to develop a new advanced ring modulation plugin that offers LFO control over modulation frequency and depth, has been successfully completed. This application is available for free download at http://www.aquaplancton.com/toby/index.html

In fulfilling the second aim of the project, "to document core developments in computer based music-oriented audio signal processing" and demonstrate how they have affected current methods, the developed plugin demonstrates the following:

Real time control over sound input, using techniques developed on from Music11 in 1973 and VAZ in 1996.

The application is not platform specific and requires no dedicated hardware. It will operate with any DAC enabled soundcard. This capability was introduced with the *.MOD format. A benefit of this attribute is that the software is free to distribute and use, again like many of the *.MOD trackers programs.

The sound generated by the plugin will be identical from computer to computer, improving upon the hardware-specific MIDI format.

The plugin works from within host based sequencers more effectively than multitasked programs like Mellosoftron 1.0. Throughout testing it has never causes a crash or audio glitch.

11 Recommendations

In investigating the Steinberg VST SDK, it has been concluded that although challenging, the task of creating one's own plugins is not insurmountable.

The task comes highly recommended as a project. Although a great amount of groundwork is necessary to complete this type of project, the results are very rewarding. An understanding of C++ is essential.

12 References

Literature references are listed by number e.g. [5]

Internet references are listed by letter e.g. [f]

12.1.1 Literature References

1. Parsons, D. "Object Oriented Programming with C++" 2nd Edition, 2000 pp.13 – 15.

2. Ince, D.C. "Software Engineering" 1994 pp 7.

3. Boulanger, R. "The CSound Book, perspectives in software synthesis, sound design, signal processing and programming", 2000, Introduction.

4. Varga, A. *disC=overy – The Journal of the Commodore Enthusiast*. Issue 2: October 1, 1996, Chapter /S07 - "Progenitor of the SID: An interview with Bob Yannes"

5. Howe, H.S. Junior, "Electronic Music Synthesis: Concepts, Facilities, Techniques", 1975, pp. 250 – 253.

6. Rasmussen, E. A. "Playing sound on a PC", *"The CWI Audio File Formats Guide"*, Version 2.10, 2002, appendix.

7. Terratec Electronic GmbH, "AudioSystem EWS64L/XL Hardware Manual" Version 1.2, 1998, pp 13.

8. Hays, T. "DirectMusic for the Masses", *Game Developer Magazine,* September, 1998, pp. 23-26.

9. Steinberg. "Virtual Studio Technology Plug-In Specification Software Development Kit", Version 2.0, 2001, pp. 5

12.1.2 Internet References

a. Paradiso, J.A. "American Innovations in Electronic Musical Instruments" 1999,

http://www.newmusicbox.org/third-person/index_oct99.html

b. Alstrup, A. "Commodore 64 Programmer's Reference Guide" 1987,

http://stud1.tuwien.ac.at/~e9426444/sidtech.html

c. Defacto2, "History of the Module", 2001,

http://www.defacto2.net/portal-music.html

d. Rindeblad, C. "The creation of the SID chip" (1998)

http://stud1.tuwien.ac.at/~e9426444/sidcreate.html

e. Digital Oracle, “Glossary”

http://www.maxreboot.com/do/Glossary/I/IBM.html

f. Hilderink, G.H. “Plug-and-Play devices in Java” (1998)

http://www.rt.el.utwente.nl/javapp/Plug-and-play/TestBeeper/

g. Boone, K. “Kevin’s computing glossary index” (2000)

http://www.kevinboone.com/compdict/frequency_modulation.html

h. “MIDI and Digital Audio Terms”, (2001)

http://www.cakewalk.com

i. “IAEKM” (International Association for Electronic Keyboard Manufacturers) (1997)

www.iaekm.org/p25.html

j. Gorham Consulting, “Thin Films Lead Storage Revolution”

http://www.goradv.com/Consulting/ConsultWhiteBriefsDS.htm

k. Net Express “Synthesizers and Wave Tables (FM, OPL2/3)” (1996)

http://www.tdl.com/~netex/sound/sound.html

l: Beseen “Hit-counter Statistics” (2002)

http://beseen5.looksmart.com/hitcounter_stats?account=1281002&counter=864572

13 Appendices

Data Sheets and Specifications

13.1.1 Transcription of Appendix O in the Commodore 64 Programmer's Reference Guide

13.1.1.1 CONCEPT

The 6581 Sound Interface Device (SID) is a single-chip, 3-voice electronic music synthesizer/sound effects generator compatible with the 65XX and similar microprocessor families. SID provides wide-range, high-resolution control of pitch (frequency), tone color (harmonic content), and dynamics (volume). Specialized control circuitry minimizes software overhead, facilitating use in arcade/home video games and low-cost musical instruments.

13.1.1.2 FEATURES

- 3 TONE OSCILLATORS
  Range: 0-4 kHz
- 4 WAVEFORMS PER OSCILLATOR
  Triangle, Sawtooth, Variable Pulse, Noise
- 3 AMPLITUDE MODULATORS
  Range: 48 dB
- 3 ENVELOPE GENERATORS
  Exponential response
  Attack Rate: 2 ms - 8 s
  Decay Rate: 6 ms - 24 s
  Sustain Level: 0 - peak volume
  Release Rate: 6 ms - 24 s
- OSCILLATOR SYNCHRONIZATION
- RING MODULATION
- PROGRAMMABLE FILTER
  Cutoff range: 30 Hz - 12 kHz
  12 dB/octave Rolloff
  Low pass, Bandpass,High pass, Notch outputs
  Variable Resonance
- MASTER VOLUME CONTROL
- 2 A/D POT INTERFACES
- RANDOM NUMBER/MODULATION GENERATOR
- EXTERNAL AUDIO INPUT

13.1.1.3 DESCRIPTION

The 6581 consists of three synthesizer "voices" which can be used independently or in conjunction with each other (or external audiosources) to create complex sounds. Each voice consists of a tone oscillator/waveform generator,an envelope generator and an amplitudemodulator.
The tone oscillator produces four waveforms at the selected frequency, with the unique harmonic content of each waveform providing simple control of tone color.

The volume dynamics of the oscillator are controlled by the amplitude modulator under the direction of the envelope generator. When triggered, the envelope generator creates an amplitude envelope with programmable rates of increasing and decreasing volume.
In addition to the three voices, a programmable filter is provided for generating complex, dynamic tone colors via subtractive synthesis.
Please note that the SID is not a FM-based synthesizer, like the Yamaha OPL series !!

SID allows the microprocessor to read the changing output of the third oscillator and third envelope generator. These outpus can be used as a source of modulation information for creating vibrato, frequency/filter sweeps and similar effects. Two A/D converters are provided for interfacing SID with potentiometers. These can be used for "paddles" in a game environment or as front panel controls in a music synthesizer. SID can process external audio signals, allowing multiple SID chips to be daisy-chained or mixed in complex polyphonic systems.

13.1.2 Gravis Ultrasound feature list

"Wavetable synthesis (16-bit instruments, up to 1 meg of on-board RAM),
stereo,
32-channel 16-bit DAC (for playing only, 16-bit recording is an option).
Sound Blaster compatible through software.
Roland MT-32 compatible through software.
8 bit ADC.
Has a 32 voice WT synthesiser.
Comes standard with 256kb of memory, upgradable to 1 Mb.
Includes MIDI/speed compensating joystick port.
Needs 16 bit ISA slot."
(Wyckoff, R. & Cornell Dobaldson, I
http://www.gamesdomain.com/pcfawq.pcfaq5.html)

13.1.3 Sound Blaster PCI128 feature list

PnP
512K Upgradeable Wave Table: 128-Voice Polyphony 128 instruments,
10 drums; 20-note/4 operator FM Synthesizer;
MPU-401 UART Midi;
Sound Samples: 1MB ROM upgrades to 28MB's.
Creative 3D Stereo Enhancement Technology for larger sound stage;
EMU8000 E-mu 3D Effects;
Reverb, chorus and pan on MIDI channels;
Includes Stereo, 16-bit, 44.1kHz Inputs and Playback;
0.5MB of Integrated Sound Font memory.
(Net Express "Synthesizers and Wave Tables (FM, OPL2/3)" (1996)
http://www.tdl.com/~netex/sound/sound.html)

13.1.4 MidiMan Audiophile 24/96 feature list

Applications Include
24-bit 96 kHz multitrack recording
MIDI recording and playback

Digital transfers; Digital mastering
LP/cassette-to-CD transfers
Computer-based Home Theater systems
Computer-based Hi-Fi systems
Specifications:
Dynamic Range: D/A 104.0dB (a-weighted), A/D 100.4dB (a-weighted)
THD: less than 0.002%
Freq. Response: 22Hz - 22kHz, -0.4,-0.4dB

13.2 Source Code

13.2.1 Source code for a VST plugin that has no effect

```
#include "AGain.hpp"
AGain::AGain(audioMasterCallback audioMaster)
	: AudioEffectX(audioMaster, 1, 1)	// 1 program, 1 parameter only
{
	fGain = 1;				// default to 0 dB (FLOAT)
	setNumInputs(2);		// stereo in
	setNumOutputs(2);		// stereo out
	setUniqueID('fork');	// identify
	canMono();				// makes sense to feed both inputs with
the same signal
	canProcessReplacing();		// supports both accumulating and replacing
output
	strcpy(programName, "Default");	// default program name
}
AGain::~AGain()
{
	// nothing to do here
}
void AGain::setProgramName(char *name)
{
	strcpy(programName, name);
}
void AGain::getProgramName(char *name)
{
	strcpy(name, programName);
}
void AGain::setParameter(long index, float value)
{
	fGain = 0.01*value;
}
float AGain::getParameter(long index)
{
	return fGain;
}
void AGain::getParameterName(long index, char *label)
{
	strcpy(label, "  Toby_Knob  ");
}
```

```
void AGain::getParameterDisplay(long index, char *text)
{
        dB2string(fGain, text);
}
void AGain::getParameterLabel(long index, char *label)
{
        strcpy(label, "   dB   ");
}
void AGain::process(float **inputs, float **outputs, long sampleFrames)
{
   float *in1  =  inputs[0];
   float *in2  =  inputs[1];
   float *out1 = outputs[0];
   float *out2 = outputs[1];

   while(--sampleFrames >= 0)
   {
      (*out1++) += (*in1++) ;    // accumulating
      (*out2++) += (*in2++) ;
   }
}
 double tremolo=0;

void AGain::processReplacing(float **inputs, float **outputs, long
 sampleFrames)
 {
    float *in1  =  inputs[0];
    float *in2  =  inputs[1];
    float *out1 = outputs[0];
    float *out2 = outputs[1];


      while(--sampleFrames >= 0)
         {

          (*out1++) = (*in1++) ;
          (*out2++) = (*in2++) ;
         }

   }
}
```

13.2.2 <u>Source code for a VST plugin that ring modulates the input with a sawtooth wave of arbitrary frequency</u>

```
#include "AGain.hpp"
AGain::AGain(audioMasterCallback audioMaster)
        : AudioEffectX(audioMaster, 1, 1)    // 1 program, 1 parameter only
{
        fGain = 1;                             // default to 0 dB (FLOAT)
        setNumInputs(2);                // stereo in
        setNumOutputs(2);               // stereo out
        setUniqueID('fork');    // identify
        canMono();                              // makes sense to feed both inputs with
the same signal
        canProcessReplacing();          // supports both accumulating and replacing
output
        strcpy(programName, "Default");         // default program name
}
AGain::~AGain()
{
        // nothing to do here
}
void AGain::setProgramName(char *name)
{
        strcpy(programName, name);
}
void AGain::getProgramName(char *name)
{
        strcpy(name, programName);
}

void AGain::setParameter(long index, float value)
{
        fGain = 0.01*value;
}
float AGain::getParameter(long index)
{
        return fGain;
}
void AGain::getParameterName(long index, char *label)
{
        strcpy(label, "  Toby_Knob  ");
}
void AGain::getParameterDisplay(long index, char *text)
{
        dB2string(fGain, text);
}
void AGain::getParameterLabel(long index, char *label)
{
        strcpy(label, "   dB   ");
}
void AGain::process(float **inputs, float **outputs, long sampleFrames)
{
   float *in1  =  inputs[0];
   float *in2  =  inputs[1];
   float *out1 = outputs[0];
   float *out2 = outputs[1];

   while(--sampleFrames >= 0)
   {
      (*out1++) += (*in1++) ;    // accumulating
      (*out2++) += (*in2++) ;
   }
}
double tremolo=0;
void AGain::processReplacing(float **inputs, float **outputs, long
 sampleFrames)
 {
    float *in1  =  inputs[0];
    float *in2  =  inputs[1];
    float *out1 = outputs[0];
    float *out2 = outputs[1];

  if ( tremolo < 0.1 )
      {
      tremolo=(tremolo+fGain);

      while(--sampleFrames >= 0)
         {

          (*out1++) = (*in1++) * tremolo *10;
          (*out2++) = (*in2++) * tremolo *10;
         }
      }

  if ( tremolo >= 0.1 )
   {
    tremolo=0;

      while(--sampleFrames >= 0)
      {
          (*out1++) = (*in1++) * tremolo;
          (*out2++) = (*in2++) * tremolo;
      }
   }
}
```

13.2.3 <u>Source code allowing user to select specific modulation frequencies.</u>

```
#include "AGain.hpp"
AGain::AGain(audioMasterCallback audioMaster)
        : AudioEffectX(audioMaster, 1, 1)    // 1 program, 1 parameter only
{
        fGain = 1;                              // default to 0 dB (FLOAT)
        setNumInputs(2);                // stereo in
        setNumOutputs(2);               // stereo out
        setUniqueID('fork');    // identify
        canMono();                              // makes sense to feed both inputs with
the same signal
```

```
	canProcessReplacing();		// supports both accumulating and replacing
output
	strcpy(programName, "Default");	// default program name
}
AGain::~AGain()
{
	// nothing to do here
}
void AGain::setProgramName(char *name)
{
	strcpy(programName, name);
}
void AGain::getProgramName(char *name)
{
	strcpy(name, programName);
}
void AGain::setParameter(long index, float value)
{
	fGain = 0.01*value;
}


float AGain::getParameter(long index)
{
	return fGain;
}
void AGain::getParameterName(long index, char *label)
{
	strcpy(label, "  Toby_Knob  ");
}
void AGain::getParameterDisplay(long index, char *text)
{
	dB2string(fGain, text);
}
void AGain::getParameterLabel(long index, char *label)
{
	strcpy(label, "   dB   ");
}
void AGain::process(float **inputs, float **outputs, long sampleFrames)
{
   float *in1  =  inputs[0];
   float *in2  =  inputs[1];
   float *out1 = outputs[0];
   float *out2 = outputs[1];

   while(--sampleFrames >= 0)
   {
      (*out1++) += (*in1++) ;    // accumulating
      (*out2++) += (*in2++) ;
   }
```

```
}
float counter = 0;
double quarterSec = 0.000011343;
double topFreq = 1000;
void AGain::processReplacing(float **inputs, float **outputs, long
 sampleFrames)
{
   float *in1  =  inputs[0];
   float *in2  =  inputs[1];
   float *out1 = outputs[0];
   float *out2 = outputs[1];
      while(--sampleFrames >= 0)
         {
                        counter = (counter + (quarterSec*(topFreq*fGain)));
                        if (counter > 0.5)
                                 {
                                 counter = 0;
                                 }
          (*out1++) = (*in1++ * counter);
          (*out2++) = (*in2++ * counter);


         }
}
```

13.2.4 <u>Source code implementing LFO modulation frequency control</u>

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "ADelay.hpp"
#include "AEffEditor.hpp"
//sets the default values
ADelayProgram::ADelayProgram ()
{
         fDelay = 0.5;
         fFeedBack = 0.5;
         fOut = 0.75;
         strcpy (name, "Init");
}
ADelay::ADelay (audioMasterCallback audioMaster)
         : AudioEffectX (audioMaster, 16, kNumParams)
{
         //sets "size" t0 44100 (1 second) - this is our delay memory
         size = 10000;
         // makes the dial make SENSE!!!!!!!!!
//       size = 44100;
         //a buffer of floats is allocated
         buffer = new float[size];
         programs = new ADelayProgram[numPrograms];
         fDelay = fFeedBack = fOut = vu = 0;
```

```
	delay = inPos = outPos = 0;
	if (programs)
		setProgram (0);
	setNumInputs (2);
	setNumOutputs (2);
	canMono();                                                    ///MAKE         IT
MONO!!!!!!!
	hasVu ();
	canProcessReplacing ();
	setUniqueID ('Kllr');                          //MAKE IT UNIQUE!!!!!

	suspend ();              // flush buffer
}
ADelay::~ADelay ()
{
	if (buffer)
		delete[] buffer;
	if (programs)
		delete[] programs;
}
void ADelay::setProgram (long program)
{
	ADelayProgram * ap = &programs[program];
	curProgram = program;
	setParameter (kDelay, ap->fDelay);
	setParameter (kFeedBack, ap->fFeedBack);
	setParameter (kOut, ap->fOut);
}
void ADelay::setDelay (float fdelay)
{
	long oi;
	fDelay = fdelay;
	delay = (long)(fdelay * (float)(size - 1));
	programs[curProgram].fDelay = fdelay;
	oi = inPos - delay;
	if (oi < 0)
		oi += size;
	outPos = oi;
}
void ADelay::setProgramName (char *name)
{
	strcpy (programs[curProgram].name, name);
}
void ADelay::getProgramName (char *name)
{
	if (!strcmp (programs[curProgram].name, "Init"))
		sprintf (name, "%s %d", programs[curProgram].name, curProgram +
1);
	else
		strcpy (name, programs[curProgram].name);
```

```
}
void ADelay::suspend ()
{
        memset (buffer, 0, size * sizeof (float));
}
float ADelay::getVu ()
{
        float cvu = vu;
        vu = 0;
        return cvu;
}
void ADelay::setParameter (long index, float value)
{
        ADelayProgram * ap = &programs[curProgram];
        switch (index)
        {
                case kDelay :    setDelay (value); break;
                case kFeedBack : fFeedBack = ap->fFeedBack = value; break;
                case kOut :      fOut = ap->fOut = value; break;
        }
        if (editor)
                editor->postUpdate ();
}
float ADelay::getParameter (long index)
{
        float v = 0;
        switch (index)
        {
                case kDelay :    v = fDelay; break;
                case kFeedBack : v = fFeedBack; break;
                case kOut :      v = fOut; break;
        }
        return v;
}
void ADelay::getParameterName (long index, char *label)
{
        switch (index)
        {
                case kDelay :    strcpy (label, " Root-Pitch  "); break;
                case kFeedBack : strcpy (label, "Speed"); break;
                case kOut :      strcpy (label, " Amount "); break;
        }
}
void ADelay::getParameterDisplay (long index, char *text)
{
        switch (index)
        {
                case kDelay :    long2string (delay, text); break;
                case kFeedBack : float2string (fFeedBack, text);      break;
                case kOut :      float2string (fOut, text); break;
```

```
          }
}
void ADelay::getParameterLabel (long index, char *label)
{
          switch (index)
          {
                    case kDelay :    strcpy (label, "- ");    break;
                    case kFeedBack : strcpy (label, " amount "); break;
                    case kOut :      strcpy (label, "   dB   ");        break;
          }
}
float counter = 0;
float countertoo = 0;
//float counterthree = 0;
double quarterSec = (0.000570162/4);
double topFreq = 10000;
double LFOFreq = 10;
void ADelay::process(float **inputs, float **outputs, long sampleFrames)
{
   float *in1  =  inputs[0];
   float *in2  =  inputs[1];
   float *out1 = outputs[0];
   float *out2 = outputs[1];
   while(--sampleFrames >= 0)
   {
      (*out1++) += (*in1++) ;    // accumulating
      (*out2++) += (*in2++) ;
   }
}
void ADelay::processReplacing(float **inputs, float **outputs, long
 sampleFrames)
{
   float *in1  =  inputs[0];
   float *in2  =  inputs[1];
   float *out1 = outputs[0];
   float *out2 = outputs[1];
      while(--sampleFrames >= 0)
          {
                              countertoo              =              (countertoo              +
(quarterSec*(LFOFreq*fFeedBack)));
                              if (countertoo > (2*3.14159))
                                        {
                                        //to cause aliases:
                                        //counter = 0;
                                        //To solve aliases:
                                        countertoo -= (2*3.14159);
                                        }
                              double nuMod = sin(countertoo);
                              counter              =              (counter              +
(quarterSec*(topFreq*fDelay)+(nuMod*(fOut/8))));
```

```
                        if (counter > (2*3.14159))
                                {
                                //to cause aliases:
                                //counter = 0;
                                //To solve aliases:
                                counter -= (2*3.14159);
                                }
                        double sinMod = sin(counter);
                        (*out1++) = ((*in1++)*(sinMod/2));
                        (*out2++) = ((*in2++)*(sinMod/2));
//                      (*out1++) = (sinMod);
//                      (*out2++) = (sinMod);
                        }
}
```

13.3 Usability Research Website Content

<u>**KillerRinger**</u>
This is a free VST plugin
Written by Toby Newman.

It will make your sounds turn funky and crazy.
Please try it out,
Distribute this URL,
and help me by sending comments to: asktoby@hotmail.com

Click here to download the VST plugin

13.4 User Feedback from Usability Research Website

13.4.1 Vesa Norilo

Well, I did. Just forgot to test it! Sorry. But here goes now.

It's neatly done, all the parameters have nice control feel with no crackling or stuff. Did you use some kind of interpolation to glide between successive parameter values? The sound is also clean and smooth.

It seems to be a ring modulator where an LFO controls the pitch of the other ring modulation source. I'm a bit mystified as to why the amount control is labeled in decibel if it is in fact the LFO amplitude as it seems to be.

Have you tried other waveforms beside sine? Square wave gives some really crunchy effects. A friend of mine has implemented bitcrusher, ring modulator and spectrum flipper in one plugin, and it sounds positively unbelievably weird :-)

Vesa

13.4.2 Igor Yael

O yes ! I've dw it , and I like it :o) I like the sound ...
very artistic sound :o)

are u thinking of a graphiX interface ?
Igor

13.4.3 Paul Stimpson

ooh i have loaded it up in orion, ooh it sounds all wibbly and metallic – wibblytastic!
i cant stop making bird noises now i've added reverb
Paul

13.4.4 Paul Random99

hi toby
yup, d/l'd your plug-in and it works well in audiomulch (a nice effect-
given the title i was half expecting a run of the mill ring modulator ;-)
Paul