

“Design and implementation of a MIDI Sampling Synthesizer”

A thesis submitted for the Degree of MEng in Electronic and Computer Engineering

Summer 2007

By Guy Burton

Supervisor: Albert Koelmans

Second Examiner: Nick Coleman

School of Electrical, Electronic and Computer Engineering

Newcastle University

Abstract

This paper is an account of the steps I have taken to design and implement a MIDI Sampling Synthesizer, starting from analyzing the existing market and looking at the general field of digital signal processing, before examining the algorithms involved, the design of a custom hardware solution and developing software based on the Steinberg VST platform to achieve the goal.

List of principal Symbols

A/D – Analog to Digital

ADSR – Attack-Decay-Sustain-Release

D/A – Digital to Analog

DFT – Discrete Fourier Transform

DLL – Dynamic Linked Library

DMA – Direct Memory Access

DSP – Digital Signal Processor

FFT – Fast Fourier Transform

FM – Frequency Modulation

FIR – Finite Impulse Response

FX – Effects

ICE – In Circuit Emulator

ICSP – In Circuit Serial Programming

IIR – Infinite Impulse Response

MIDI – Musical Instrument Digital Interface

PCB – Printed Circuit Board

SPI – Serial Peripheral Interface

SRAM - Static Random Access Memory

SSI – Synchronous Serial Interface

STDFT – Short Time Discrete Fourier Transform

VST – Virtual Studio Technology

Contents

1. INTRODUCTION	1
1.1 DETAILED PROJECT AIMS:.....	2
2. LITERATURE REVIEW	2
2.1 OVERVIEW OF SOUND SYNTHESIS TECHNIQUES	2
2.2 BRIEF HISTORY OF SAMPLE BASED SYNTHESIS:	3
2.3 AUDIO CONVERSION AND INTERFACING	6
2.4 MULTI RATE SAMPLING	6
2.5 PITCH SHIFT ALGORITHMS	8
2.6 EFFECTS	11
2.7 MIDI INTERFACING.....	12
2.8 CONCLUSIONS	12
2.8.1 Data Structures.....	13
3. IMPLEMENTATION OF AUDIO ALGORITHMS	14
3.1 FILTERING	14
3.2 FAST FOURIER TRANSFORM	16
3.3 RE-SAMPLING AND PITCH SHIFTING	16
3.4 LIBRARY CODE	17
4. REAL TIME CONSIDERATIONS.....	17
4.1 SCHEDULING AND REAL TIME SYSTEM CONSIDERATIONS	17
4.2 DIGITAL SIGNAL PROCESSORS AND MICROPROCESSORS.....	18
4.3 PLATFORM CHOICE	20
4.4 ANALYSIS OF MEMORY AND PROCESSING REQUIREMENTS	21
5. HARDWARE DESIGN	22
5.1 SCHEMATIC DESIGN	22
5.2 PCB DESIGN	24
5.3 ASSEMBLY AND MANUFACTURE OF HARDWARE.....	25
5.4 TESTING OF HARDWARE	25
6. CHAPTER – SOFTWARE DESIGN	26
6.1 STEINBERG VST IMPLEMENTATION	26
6.2 FX CODE.....	28
6.3 CUSTOM HARDWARE DEVELOPMENT.....	29
7. RESULTS	29
8 DISCUSSION	31
8.1 DEVELOPMENTS FROM THE PROJECT	33
8.2 FUTURE OF DSP HARDWARE IN AUDIO SYNTHESIS	36
9. CONCLUSIONS.....	37
10. REFERENCES	37
11. APPENDICES.....	41
APPENDIX 1 - TEST CRITERIA.....	41
APPENDIX 2 - MENU STRUCTURE.....	42

APPENDIX 3 – BILL OF MATERIALS	43
APPENDIX 4 - LIST OF FILES	44

Table of Figures

Figure 1 - Gantt Chart	
Figure 2 – Re-sampled Signal by ratio of 4/3	
Figure 3 - Polyphase re-sampled signal by ratio of 4/3	
Figure 4 - Compression and Expansion of the time axis using TDHS (taken from (30))	
Figure 5 - Delay effect structure, taken from (32)	
Figure 6 - Reverberation Impulse Response taken from (33)	
Figure 7 - Data Structures	
Figure 8 - Analog Devices SHARC 21261 internal architecture	
Figure 9 - System architecture	
Figure 10 – Schematics	
Figure 11 – Memory Interfacing Schematics	
Figure 12 – Audio Codec Schematics	
Figure 13 – External Connections	
Figure 14 - PCB design	
Figure 15 - 3D impression of assembled PCB	
Figure 16 - Photo of assembled main PCB	
Figure 17 - Photo of interface board	
Figure 18 - Class Diagrams	
Figure 19 - Screenshot of VST Parameter Editor	
Figure 20 - Screenshot of VST GUI	
Figure 21 - Screenshot from Waves API 2500 Compressor VST Emulation	
Figure 22 - Photograph of API 2500 Hardware Compressor	

1. Introduction

In this thesis I will be presenting the steps involved in my design of a MIDI Sampling Synthesizer in both hardware and software. The purpose of this paper is to investigate how to optimize the design of a low cost sampler using design compromises and best practice derived from published research in the field and commercial sources, and assess its performance. The hardware project was not successfully completed; however the software implementation achieved the aims set out in the form of a virtual instrument for a VST software host on a personal computer.

I first provide an overview of sound synthesis and sound effect design and also to the process of digital signal sampling and processing as well as MIDI protocol. A description of the data structures used in the project to represent basic units of a sampling synthesis system follows this. The first chapter of the main text discusses the high level implementation of audio algorithms in terms of processing, quality and memory tradeoffs rather than coding details. As a result of this analysis I then go on to consider the requirements of a sampler as a real-time system, and in terms of processing power. This section also discusses the benefits of a digital signal processor over a general purpose microprocessor, and the process of selecting a development platform. I then move on to discuss the design of the custom hardware. The following chapter covers the software design for the system, though this does not cover the algorithms which were detailed in the first chapter, focusing more on the structure and API's used in the code. The results section regards the output of the sampler implementation in terms of quality and performance, this section is largely qualitative analysis.

It is important to note the clash of terminology between the musical sense of the word 'sampling', and the meaning given by the field of digital signal processing. Musically speaking sampling is the general process of recording audio and using it as the basis for synthesis. In engineering however it has a specific meaning of converting a continuous signal into a discrete value sequence. Hopefully the usage will not be too ambiguous during this thesis. What I will refer to as waveforms, meaning a length of real values representing an audio signal, are commonly know as 'samples' in the music industry, but I will use the word 'sample' to mean an individual real value of a waveform.

See Figure 1, a Gantt chart detailing the estimated time scale of the project and the tasks which I have broken it down into.

1.1 Detailed Project Aims:

To implement a polyphonic 16 bit 44.1 kHz sampling synthesizer with the following features:

- Translate MIDI On and Off messages
- Polyphonic operation
- Sustain looping
- Amplitude and Filter ADSR (Attack, Decay, Sustain, Release) Envelopes
- Multiple Samples per patch with individual key ranges and base pitches
- Some basic effect implementations with editable parameters

2. Literature Review

2.1 Overview of sound synthesis techniques

“There are two basic approaches to synthesizing musical sound. “Direct Synthesis” involves creating a sound purely from mathematics. Signals from oscillators, envelopes, and filters are combined to mimic the timbres and contours of the desired instrument. Envelopes provide a contour and are used to control the amplitude and other slowly changing aspects of a sound. Filters can change the spectrum of a sound. Direct Synthesis works well for many instruments. It is difficult, however, to match the sound of complex instruments with a few oscillators and filters. So another technique called “Wavetable Synthesis” was invented. Wavetables are digital audio recordings of the actual instrument that serve as the basis for the synthesized sound. This wavetable recording can then be shaped with envelopes and filters to create the final sound.”

- P. Burk [1]

Roads [2] describes additive synthesis as “the summation of elementary waveforms to create a more complex waveform”. This form of synthesis is one of the oldest, based upon the Fourier series, and is seen in the mechanical Telharmonium, and the Hammond ToneWheel Organ. He continues, “Subtractive Synthesis implies the use of filters to shape the spectrum of a source sound”. This is the most common form of synthesis, whether it is performed with functional or wavetable based oscillators, and can be combined with additive synthesis.

“Fixed-waveform table-lookup synthesis”, or wavetable synthesis, is “the process of repeatedly scanning through a lookup table in memory containing one period of a waveform” [2]. Massie describes “Sampling Wavetable Synthesis” as “playback of digitized audio waveforms from RAM wavetables, combined with sample rate conversion to provide pitch shifting” [3]

Trautmann and Cheung [4] consider the allocation of processing resources in real time for wavetable instrument synthesis. They state that “although wavetable synthesis is a single paradigm of sound production, there are a number of factors which affect resource usage. These include the type of interpolation, type of filter, and which parameters are updated”. This study offsets the ease of computation involved with FM (Frequency Modulation, a popular technique due to its computational simplicity) synthesis against the realism of sampling synthesis. Massie [3] describes a similar trade off between cost, accuracy, and expressivity. “Sampling has high accuracy, but weak expressivity, where [direct] Synthesis has high expressivity and weak accuracy. Both technologies are evolving towards an ideal with both high accuracy and expressivity, but at a higher cost for implementation”.

Massie [3] then gives a list of areas to consider in the capabilities of a “full featured sampling synthesizer” which I have reflected in the project aims:

- Audio playback
- Pitch shifting technologies
- Looping of sustain
- Enveloping
- Digital Filtering
- Amplitude Variations as a function of velocity
- Mixing or summation of channels
- Multiplexed Wavetables

2.2 Brief History of Sample Based Synthesis:

The following information was taken from [5]. The Fairlight CMI, released in 1979, was the first commercial product available which could perform sample based synthesis. It was based upon two 8-bit Motorola 6800 processors and 8 16K DRAM chips which at the time were very expensive, accounting for the \$25000 retail price for the system. It had many

revolutionary features other than sampling, but it was this that caught the eye of E-mu systems, who proceeded to develop the Emulator, which was an attempt to reduce the cost of a sampling system (to a mere \$8000) in order to bring it to the mass market. This machine, produced from 1981 to 1984, was based upon the general purpose Z-80 microprocessor, and it contained five DMA (direct memory access) chips. This was still very expensive and only offered 8-bit conversion and 8 note polyphony; hence only 500 units were sold.

Ake and George [6] describe the design of a microprocessor based digital sampler based upon the Motorola 68000 microprocessor. They conclude that the key factors in the design of such a system are memory size/type, sampling rate, word size and choice of microprocessor. These should be chosen based on the appropriate compromises of cost, development time, and the specification of the final product.

In 1986 there was a revolution as several cheaper products, such as the E-mu Mirage and the Casio FZ series, hit the market. Amongst these Akai released the 12-bit S900 at around \$2500:

“The Akai S900 was the first sampler that brought affordable, studio quality sampling in an easy to use and flexible rack mounting format to many musicians and studios the world over becoming an industry standard by which all other samplers were compared.” [7]

Whilst sampling was available to studios and professional musicians at this point, it took a lot longer for wavetable synthesis to take over from FM in the home/consumer market, in the form of 16 bit Industry Standard Architecture (ISA) sound cards for personal computers, for example the SoundBlaster AWE32. Several dedicated DSP chips (which in this case meant Digital *Sound* Processor) were developed, which provided all of the rudimentary functions for wavetable synthesis. For example [8] describes the development and architecture of the SWS-63 wavetable synthesizer integrated circuit, and [9] describes the architecture of an optimized 32 voice 16 bit wavetable synthesis integrated circuit, using a double pipeline to calculate the two key things for sample synthesis: sample address and sample calculation.

Commercial examples of single chip DSPs are the Samsung KS0164, E-mu Proteus, and Ensoniq ES-5506. The feature set of the latter is described at [10]. The E-mu 8000 was the device used on the AWE32 soundcard. “The EMU-8000 has 32 individual oscillators, each playing back at 44.1 kHz. Each oscillator consists of a resonant low pass filter, two LFOs

(low frequency oscillators), and two envelope generators.” [11]. [12] describes the Turtle Beach MultiSound Classic, which was a powerful ISA sound card incorporating a 68000 microprocessor, a DSP (digital signal processor) chip, the E-mu Proteus wavetable synthesis chip, a large amount of RAM, and 8MB of wavetables on ROM.

There has been a marked shift in recent years toward soft-samplers, for example Steinberg HALion, E-Mu Emulator X2, and Tascam Gigastudio. These run either as standalone applications or as Steinberg’s VST instrument (VSTi) or DirectX instrument (DXi) plug-ins within a DAW (Digital Audio Workstation) host application on desktop computers (either PC or Apple Mac). This is due to the drop in price of personal computing hardware, and the increase in RAM and processor power. There are several “open ended” hardware sampling systems such as the Roland VariOS, which run software editable on a PC but offload the bulk of the processing to an external DSP system to reduce CPU overhead.

Steinberg Virtual Studio Technology (VST) is an open standard protocol for integration of virtual effect processors and instruments into the digital audio environment. Steinberg provide a Software Development Kit (SDK) which contains a set of base classes, which provide a host application with an encapsulated interface for block based audio streaming controlled by MIDI, with a set of parameters which may or may not have a corresponding Graphical User Interface (GUI). There is a separate development kit for creating VST GUI’s, which means that the audio processing code can be kept separate from any interface code which may be platform specific [13].

“VariOS is an open-ended hardware/software system that allows musicians to build audio-based tracks with the same flexibility as MIDI. All processing is handled by the VariOS module, which is controlled using bundled software. The first software bundle is V-Producer for VariOS. This software lets musicians manipulate a sample's pitch, time and formant, add effects and even build complete arrangements while keeping the music flowing and without CPU drain. And since the VariOS module can be updated, the system can take on entirely new functions in the future.” [14] This statement from Roland describes the vast possibilities for the product.

2.3 Audio Conversion and Interfacing

In order to convert to and from the digital domain, PCM (pulse code modulation) is used. This is the process of filtering, sampling, quantizing and encoding an audio stream as a binary word [15]. Analogue to digital (A/D) and digital to analogue (D/A) converters are required, as well as buffer amplifiers, anti aliasing and smoothing filters. All of these functions however can be provided by one audio codec chip as described in [16], via a Synchronous Serial Interface (SSI).

A general purpose microprocessor may not be powerful enough to perform the necessary sample fetch operations, I/O, and signal processing for a sampler in real time. A large amount of multiply and-add operations are required for digital signal processing, especially for algorithms which use time domain convolution or require conversion between time and frequency domains via a FFT (Fast Fourier Transform) algorithm. This means that there are many optimizations to the standard CPU architecture that can be made to form a dedicated DSP chip. A fully parallel multiplier is used within the ALU (Arithmetic Logic Unit), capable of producing a result within one clock cycle. This is placed directly within the data path of the ALU, to immediately produce a result, and immediately followed by an adder, a technique known as pipelining [17].

Traditionally in real time audio systems a conventional microprocessor would be used for interfacing tasks and a DSP would be used solely in a signal processing role. The LucasFilm Digital Audio Facility was an early example (1985) of this, using several MC68000 microprocessors to interface with video screens and digital mixing console controls, whilst leaving the audio signal processing to a bank of up to eight DSP chips. This provided extremely high data throughput of up to 72 channels of 24 bit 48 kHz audio. [18] However multiprocessor development would not be feasible within the time frame for this project, and there should be enough spare processing capacity to handle simple I/O for this system.

2.4 Multi Rate Sampling

The algorithm which is most fundamental to the concept of a sampler is that of sample rate conversion. When a note is played which differs in pitch to the waveform stored in memory, it must be re-sampled to provide a note of the desired frequency. A note which is higher than the stored waveform must be decimated or down sampled; where as a note which is lower must be up sampled or interpolated. Both of these operations cause problems with the

frequency spectrum. Both interpolation and decimation algorithms can only deal with integer values, so re sampling by a real number must be expressed as a fraction L/N which relates to an interpolation of L times followed by decimation by N [19].

Interpolation is at its simplest level, a zero order hold function, which repeats the last output sample until it is ready to move onto the next sample in the wavetable. An alternative to this is two step linear interpolation which uses two adjacent samples and a decimal weighting between them. The frequency response of each of these two methods is compared in [3]. This article also considers the efficiency of more complex interpolation techniques and the implementation of an FIR “imaging” filter to remove false frequencies created by the sample interpolation.

Fitzgerald and Anderson [20] analyze the spectral distortion when using the zero hold approach and conclude that each discrete spectral component in the original time series splits into N discrete spectral components described by $\omega_m = 2\pi m / LT$ for the frequency shift from the original spectral component, where their amplitudes D_m are described by the Bessel Function table.

The optimal approach for interpolation by L takes each sample from the waveform and pads it with $L-1$ zeroes before the next. This produces a waveform of L times the length of the original. The frequency spectrum of this signal is then equal to original frequency spectrum replicated at each multiple of F_s . This is because the process of adding zeroes is equivalent to multiplication of each sample of the waveform by the following vector $\{1, 0, 0, \dots, 0_{L-1}\}$. In the frequency domain this is convolution of the waveform with a comb function. An anti-aliasing filter after the interpolation is required to remove this effect. The filter cut off required is equal to half of the original sampling rate. As the energy from the original waveform is spread over L times as many samples, a gain of L is required after the filter [21].

Decimation causes imaging to occur since even when samples are dropped, all the frequencies present in the original waveform will still be present at some point in the spectrum. To prevent this, an anti imaging filter must be used before down sampling takes place. This takes the form of a low pass filter, with a cut off value equal to the new Nyquist limit, or half of the new sampling rate.

When converting by a rational value L/N , thus implying integer interpolation by L followed by integer decimation by N , the two filters used are both of the low pass variety and so can be

combined by taking the most stringent of the parameters. However, interpolation followed by the combined filter and then decimation is still a computationally expensive process, because a large portion of memory is used and a lot of multiplies with zeroes are unnecessary. An optimised method of re-sampling is to use polyphase filters. This is discussed by Massie [3] in the context of a wavetable sampler, and Crochiere and Rabiner [21] who provide an in depth analysis into methods of optimizing interpolation and decimation schemes as well as the memory/processing trade offs available. Figure 2 shows the process of re-sampling by a rational factor of $4/3$. This was performed using non optimised code. It uses a large amount of memory to generate the re-sampled waveform. Figure 3 shows the output of a polyphase implementation of rational factor re-sampling. The only additional storage required is the impulse response, and it also uses far fewer computations. Note that both figures are output shifted to compensate for the latency or phase shift caused by the FIR filters.

I have discussed re-sampling as a method for pitch shifting of a wave form. This is the normal *modus operandi* for a sampler. However it is not ideal because when pitching up certain waveforms, because the duration of the note played is reduced. Typically a musical note with non-trivial harmonic information will be considerably reduced in quality after a shift of even an octave. This is not necessarily a quantitative measure of the signal so much as a heuristic determination of perceived “realism” of the musical note produced after the shift. A vocal sample, for example, sounds ridiculous after re-sampling by only half an octave up for two major reasons. Firstly, the waveform is faster, so the phrase will sound unnatural for spoken word. Secondly the formant pitch of the voice, being normally static regardless of the pitch produced by the vocal tract, will be raised.

However, for the majority of sampled instruments, re-sampling provides an adequate solution, and is useful for other functions. For example the tempo drum beat can be altered and looped to fit the music. If a range of greater than an octave shift is required to produce realistic reproduction of an instrument, several differently pitched waveforms can be used in a patch. Each sample can be assigned to a different range of the keyboard and thus reduce the amount of pitch shift required. This is a technique called multisampling, and is a tradeoff between memory and output quality.

2.5 Pitch Shift Algorithms

An alternative approach to re-sampling is available in the form of asynchronous pitch shifting. This is an umbrella term for taking a waveform and modifying either its length or

pitch whilst leaving the other variable unchanged. It is valuable to note that there is a difference between pitch shifting and frequency shifting. The latter is taking the frequency spectrum and moving it by a given amount. This is not particularly useful for musical purposes because most signals which we would like to sample have several harmonic frequencies because we are using the waveforms as a basis for subtractive synthesis. When frequency shifting occurs the harmonic relationships are not preserved, and thus the musical qualities are lost. For example a signal with $f_0 = 1$ kHz and $f_1 = 2$ kHz shifted up by one octave produces $f_0 = 2$ kHz and $f_1 = 3$ kHz. The first harmonic is no longer related to the fundamental frequency.

Single-sideband Frequency Modulation (SSBFM) provides a method for frequency shifting. “This process entails eliminating the negative frequency content of a signal, modulating the positive frequencies by multiplication with a complex sinusoid, and finally reconstructing the real signal.” [22] The same article concludes that with this method there is a significant qualitative degradation of sound quality. “The original warm sound of the music becomes metallic and dissonant. While the pitch is audibly higher, the shift introduces significant harmonic distortion.”

The Phase Vocoder technique is a popular technique for pitch shifting, developed by Flanagan and Golden [23] using filter banks, and later by Portnoff [24] implemented using the FFT algorithm. It is a method of analysis and synthesis of a signal which is an identity function when no pitch shift parameters are changed. Moorer [25] discusses the Phase Vocoder and its applications for music and concludes that whilst it is a mathematically sound principle, “when pitch changes are made by blindly multiplying all frequency traces by a fixed factor, we sometimes get a strange ‘choral’ effect. Although this may be useful, it is not what we expected, and we seek some way to control this effect”. He provides several graphs showing that the frequency-time plot of a human voice, especially the higher harmonics, are quite non-periodic, which reduces the effectiveness of the technique. Similar graphs for a saxophone however show high degrees of periodicity, and thus for our purposes of musical synthesis may be valuable.

Essentially the algorithm breaks the signal into its constituent sinusoidal components, multiplies their frequencies by the pitch shift ratio, and re-synthesizes the time domain signal. A time varying signal must be broken into frequency static sections of length N . This also helps the algorithm perform in real time, as long length DFT’s are inefficient and cause large

latency. The output of the DFT consists of $N/2+1$ complex values, containing magnitude and phase information. In order to be useful for the Phase Vocoder, these must be converted into magnitude and frequency components. Bernsee [26] explains how this is done by comparing the phase offset of a sinusoid over a series of frames to find the deviation of the signal from the bin frequency in which it resides.

In order to improve the quality of the DFT output, a window function must be used. Puckette and Brown [27] provide an analysis of the accuracy of the frequency estimates in the Phase Vocoder technique, showing the predicted RMS error for varying frame overlaps for several window functions. This is also discussed in the context of the Phase Vocoder by Abeysekera and Padhi [28]. This paper uses the Cramer-Rao bound (CRB) to compare the performances of several windows with the rectangular window function for minimizing the error in frequency estimation of a signal comprising either of a sinusoid and white noise at varying signal to noise ratios, or a multiple sinusoid signal. It also analyses the effects of varying window lengths and the effect of overlap on frequency estimation.

In order to correctly re-synthesize the signal after Phase Vocoder analysis, frames must typically overlap by 75%. This incurs a large amount of computation. Laroche and Dalton [29] present a method for reducing this to 50% in the situation where only integer frequency shifts (multiples of N/T_s) are required.

Time Domain Harmonic Scaling (TDHS) [30] is a technique shown in Figure 4 for real time pitch modification of speech, originally implemented in 1983 on a very low (by today's standards) power DSP chip. It allows compression and expansion of the time axis of a signal by a process of pitch detection based upon using the auto correlation function (ACF). This is followed by an algorithm which for compression merges two periods of speech by using a saw tooth window on the first block of samples and its inverse on the next block, and for expansion uses a saw tooth window over two periods starting at N and its inverse over two periods starting at $N+1$. This means that blocks line up without mismatch.

Whilst it is proven to be effective for speech, the quality of its output depends highly on the ability for the algorithm to detect the pitch of the waveform. The quality of pitch detection is discussed in the paper. For a simple musical signal such as a trumpet, piano etc, the algorithm should be fairly capable. For speech, a low pass filter around 1 kHz is suitable to remove second formants and other pitches which may interfere with detection of the fundamental.

However a musical note could have a fundamental of a much higher pitch than this, which means either simply a higher frequency or a user adjustable cut off must be used. Both of these options are not very reliable. For a more complex musical signal such as a chord or short phrase it would struggle.

S. Bernsee [31] offers the following summary: “It is very difficult to objectively rate or compare various time compression/expansion and pitch shifting algorithms with regard to quality due to their nonlinearity and signal dependency. It is highly difficult to establish a solid measure to estimate their overall performance from simple test signals, because most of them tend to do very well with test signals due to their simple structure”

2.6 Effects

“Delay is one of the simplest effects out there, but it is very valuable when used properly.” [32] Delay is simple to implement in a digital system with two parameters: feedback gain and delay mix, as shown in the Figure 5. A simple circular buffer is used to store the last x outputs and each new output is calculated by removing the last value from the buffer and multiplying by the mix coefficient. The last value is then multiplied by the feedback coefficient and added to the next sample going into the buffer.

“Reverberation (reverb for short) is probably one of the most heavily used effects in music. Reverberation is the result of the many reflections of a sound that occur in a room.” [33] Figure 6 describes the typical impulse response of a room in the time domain. Gardener [34] states the following: “We can simulate the effect of the room by convolving an input signal with the BIR (binaural impulse response); where h_L and h_R are the system impulse responses for the left and right ear respectively, $x(t)$ is the sound source, and $y_L(t)$ and $y_R(t)$ are the resultant sound signals.” This approach means that we can use an FIR (finite impulse response) digital filter algorithm to calculate the reverb. There are two typical approaches to digital reverb synthesis. One is to record an impulse response in a room or other space and perform a convolution on this. The other is to simulate the room using an algorithm fed by a set of parameters including room type, room size, decay length, and high frequency suppression to determine the impulse response.

Equalization (EQ) can be used as a corrective or creative effect in synthesis, either in cutting or boosting frequency bands in order to adjust the overall balance of the sound. EQ effects

are typically one of three types: fixed band, parametric, or graphic. Fixed band equalizers are the simplest, providing a set (typically three: low, mid and high) of fixed frequencies and corresponding gain controls. Graphic equalizers are similar, though offering many more bands (between 15 and 31). Parametric equalizers offer the most control, offering continuously variable frequency and gain as well as a bandwidth control for each band. A typical parametric equalizer may offer two bands switchable between shelving EQ and high/low cut filter, and two bands described above which are essentially band-pass filters.

2.7 MIDI Interfacing

“MIDI (Musical Instrument Digital Interface) is an industry-standard protocol that enables electronic musical instruments, computers and other equipment to communicate, control and synchronize with each other.” [35]

The standard has been maintained by the MMA (MIDI Manufacturers Association) since 1985. “The original Musical Instrument Digital Interface (MIDI) specification defined a *physical connector* and *message format* for connecting devices and controlling them in real time.” [36]

MIDI information is transferred over a 31 kbps baud asynchronous serial connection as one or two byte messages such as “note on”, “note off”, “patch change” etc. The sampler will use MIDI to trigger notes to be played.

There are several tuning systems in western music which translate the note names and thus their corresponding MIDI numberings to frequencies. The most common of these is called Equal Temperament and it is the most simple in that the ratio between the frequencies of two adjacent notes is always the same. The following equation, used throughout this project, relates a note n 's frequency P_n to a reference note a at frequency P_a . [37]

$$P_n = P_a \times 2^{n-a/12}$$

2.8 Conclusions

The development process for this project will follow the traditional waterfall (linear) engineering process as described by Ackenhusen [17]. It is important for the project to remain within budget and time constraints, so a disciplined approach is required. For the hardware design, the high level blocks or modules for the system will be decided upon,

before choosing specific integrated circuits or components. After the design is completed a timing check will be required to make sure that each component is compatible. At this stage the PCB will be designed. The circuit will be fabricated concurrently with specification and development of the DSP functions which will be used.

In order to test the system, a list of high level tests has been written based on the specification. This can be found in appendix 1. These tests will be performed at the end of development to ensure the quality of the final product. Unit testing will also be used during the software development process. For each software module, tests will be devised before the development commences. This is a very suitable development model for a single developer project as it allows rapid changes to the specification and structure of the code. [38] [39]

2.8.1 Data Structures

After completing research into synthesis techniques and technology, I have come up with diagrams to show the data structures required to store and process audio data. See Figure 7 for class diagrams in the style of Unified Modelling Language (UML). The highest level data structure is Sampler. This stores all of the Waves and Patches and contains state information.

The Wave class contains pointers to a waveform's storage location in memory and the following parameters: name, volume, waveform length in samples, sample rate, start point, end point, bit depth, and oversampling rate.

The Patch class represents a 'playable' entity. A Patch is composed of several KeyZone objects, each of which contains a reference a waveform in memory, and the following parameters: Filter envelope, amplitude envelope, volume, active MIDI key range, loop points, original pitch, and fixed pitch mode. These parameters affect the way in which the sample reference is processed at the point of playback. Combining several KeyZone objects within a Patch allows "layering" of sounds. Note that a sample may be referenced by multiple KeyZone objects, as all processing occurs in real time at the point of playback and is non destructive.

The FX class is a base for time and frequency domain effects. It contains pure virtual methods for the menu functions to display and edit three parameters. This is an arbitrarily small number for simplicity. A more advanced approach would be to have a method requesting the number of parameters from the FX object and use this to access a function pointer array for the increment and decrement functions. Each patch has a field containing a

TimeDomainFX, a sub-class of FX which operates on a sample by sample basis. Each KeyZone has a FrequencyDomainFX object, a subclass of FX operating on blocks of frequency domain data.

3. Implementation of Audio Algorithms

3.1 Filtering

Subtractive synthesis involves taking a complex waveform such as a harmonically rich function such as a square wave and using filters to shape the sound in the desired way. Sampling synthesis is similar to this, only using a recorded waveform as the basis for filtering. Thus the amount and type of filters provided by the sampler are a large part of the flexibility in the synthesis options available.

The overall filter for the patch should be a time domain algorithm, since the patch sums the samples from each of the key zones as they are output. Analog filters are widely regarded to sound more ‘musical’ than their digital counter parts. For this reason an IIR (Infinite Impulse Response) filter will be used. The advantage of this type of digital filter is that it can model or be derived from an analog filter transfer function. They also require fewer coefficients and thus less computation and memory than the other type of digital filter, the FIR (Finite Impulse Response) filter. The drawbacks of an IIR are that it uses feedback of the filtered signal, which means that any inaccuracy due to coefficient quantization or finite word lengths can cause instability in the output. These effects can be reduced by using floating point arithmetic and the choice of implementation structure.

A Butterworth filter is a good choice for the IIR model, since it has a monotonic pass band. This is very important as it means there will not be any distortion below the cut off point. The drawbacks of this type of filter are that there is non linear phase shift around the cut off point and that the cut off “knee” is rounded. Both of these negatives however are acceptable within the context of the sampler. The coefficients for the Butterworth filter are relatively easy to compute compared to other types of filter. The filter coefficients must be calculated for each value of the filter cutoff, which means that this filter would require much more computation if it were to be modulated by an envelope or LFO (Low Frequency Oscillator).

Each of the key zones within a patch has a filter which is modulated by a time ADSR envelope. This envelope controls the filter cut off value. If the waveform is stored within a

patch in the frequency domain, the filter can be implemented as a simple multiply function with the desired frequency response. This requires very little computation time and can produce very extreme frequency responses which would not be possible with an analog filter. A simple implementation for this is a filter function with a simple two line model with a 12dB/octave roll off. The frequency response for a desirable analogue filter could be stored in a lookup table or calculated in real time from its transfer function if desired, depending on the tradeoff between processing power and memory. This would produce a more natural and thus 'musical' filter.

This method of filter implementation is known as Fast Convolution. Time domain FIR implementation involves a convolution with the desired filter impulse response (obtained from the IFT of the frequency response) which corresponds to an $O(N^2)$ function, whereas performing an FFT and consequently a frequency domain multiplication reduces the filter function to $O(N\log_2N)$ where N is the filter coefficient length.

The drawback of performing the filter operation in the frequency domain for the sampler is that it sets the minimum resolution of the filter envelope to the size of the FFT frame. This is because in the frequency domain we lose temporal placement information and so the filter cut off can not be incremented on a sample by sample basis. On the other hand, the time domain FIR coefficients would have to be recalculated for each value of the filter cutoff, and thus for virtually each sample (or for the desired envelope resolution), which is a relatively intensive computational process. There are two main ways of calculating filter coefficients. The first is the windowed sinc method, which uses the filter specification to choose a window function and thus defining the number of coefficients required. These are then calculated by multiplying the window function with the sinc function. E.g. for the n^{th} coefficient the following formula would be used using a Hamming window:

$$W[n] = 0.54 + 0.46 \cos(2\pi \cdot n)$$

$$Y[n] = \sin(\pi \cdot n \cdot \Omega_c) / (n\pi) * w[n]$$

The second approach is to use the Parks-McClellan algorithm which uses the Remez exchange algorithm to find the optimum linear phase filter coefficients given the specification in terms of the worst case error of the output signal [40]. This is the common approach taken as it produces the best coefficients possible for the specifications.

3.2 Fast Fourier Transform

Fast Fourier Transform is a blanket term for optimized DFT algorithms. The Cooley-Tukey algorithm uses a divide and conquer approach to reduce the large transform into smaller chunks, decimation in time radix-2 for example recursively splits the input into pairs of data which are trivial to process. The DFT performs a complex to complex mapping, however as audio is purely real valued data; there are optimized FFT routines which can be used which can save roughly a factor of two in processing and memory usage. The length of the transform window is a tradeoff between frequency resolution and time resolution. Most waveforms for a sampler will have relatively stationary frequency content so time resolution or the loss of temporal placement is not a big issue. A window function should be used on the transform in order to reduce spectral leakage. This occurs when a continuous frequency component does not fit exactly into a discrete frequency bin and thus has its energy smeared across surrounding frequency bins. This leakage follows the shape of the FT of the window function used, so for a rectangular window (no window), this corresponds to the sinc(x) function. A window can be chosen which spreads the energy across the bins in a different manner more conducive to the analysis or algorithms to be used. Zero padding is a technique which involves adding a series of zeroes to the end of the sample data. This increases the transform size without reducing temporal resolution and thus reduces the width of each frequency bin, which may aid the accuracy of frequency analysis, at the cost of greater computation time for the longer transform.

3.3 Re-sampling and Pitch Shifting

Polyphase filtering is a technique for rational value sample rate conversion which removes the unnecessary intermediate interpolation step which uses memory and processing operations. Since decimation means that only one in N filtered samples are output, it seems unnecessary to calculate N-1 intermediate values. This can be avoided by splitting the filter $h[n]$ into L separate sub-filters, and evaluating the output of these in turn for consecutive samples, producing the re-sampled output.

The output is evaluated as the sum of the sub-filter outputs:

$$y[m] = \sum_{n=-W/2}^{W/2} g_m(n) \cdot x[\text{floor}(mM/L) - n]$$

where $g_m(n) = h(nL + (mM \bmod L))$ for all m output samples and filter coefficients n.

The Smith and Gosset method is similar to this but more commonly used for re-sampling in audio synthesizers. Whereas polyphase filters operate on separate sub-filters created from the reordered interpolation filter function, the Smith and Gosset method selects the coefficients from the filter table at run time. This allows greater flexibility for varying the conversion ratio which is common for synthesis functions such as oscillator LFO's, pitch bend and pitch modulation controller.

3.4 Library Code

There are a number of free C/C++ libraries available which implement various DSP functions. The DIRAC LE library contains functions for asynchronous pitch shifting and time stretching. It is released on a free license as a cut down version of a commercial library, which also offers features such as formant correction, pitch correction and sample rate conversion. Secret Rabbit Code provides libamplerate which is an open source sample rate conversion library, providing high quality conversion by real ratios. libdsp is a library of DSP functions such as filtering, transforms and polyphase re-sampling on which development has now ceased. It is only available for Linux and Solaris platforms, although the source code is available for porting. AIFF and WAV are the two most common wave formats for PC and Apple Mac. The former was developed by Apple Computer, and the latter by Microsoft. They both use the Resource Interchange File Format, which is a generic meta-format for storing data in tagged chunks [41]. I have used the MiniAiff library, developed by Stephen Bernsee, which provides a very simple API for loading AIFF files.

4. Real Time Considerations

4.1 Scheduling and real time system considerations

A real time system has hard and soft time deadlines which are specified. An audio synthesizer must operate in real time because each sample from the wave memory must be processed and delivered to the output at regular intervals. There are no truly hard deadlines in that nothing disastrous will occur if a few samples are dropped, although in a performance this is not desirable. To guarantee real-time performance, feasibility checking is required. This uses algorithms to determine whether or not it is possible to schedule a given set of periodic instructions based on their specified processing durations. If the feasibility check passes then the system can execute correctly and complete before all deadlines. If it fails however, then tasks may need to be cancelled based on a task priority system. In an audio synthesizer this could be useful in that if the check fails at any point then the quality of output

could be lowered by changing algorithms used or not performing certain operations in order to meet the output deadlines. Real time scheduling in this way is quite complex to implement and requires quite accurate estimates of processing time in order to be useful. It also only works effectively when the scheduler is in total control, and thus there is no underlying operating system or other threads of execution. Garbage collection of memory in managed programming languages also causes problems for scheduling algorithms. When running a program on a proprietary hardware device it is possible to calculate exactly how many clock cycles a given method will take, and there will not be any other code executing so the scheduler can be in total control. A feasibility check could be performed on each MIDI note on instruction to determine if there are enough resources to handle the required processing.

Because of the nature of frequency domain algorithms used, block based processing is necessary. Ackenhusen [17] states that “The threshold of $T_c = LT_x$ is a vital one to real time signal processing (where T_c is the computation delay, L is the frame length and T_x is the period of the signal). Systems that have $T_c < LT_x$ are said to operate in real time, those that have $T_c > LT_x$ operate in non real time.” In all block based solutions there is a delay between the first input (in this case the MIDI note on message) and the first output (in this case the first processed audio sample reaching the D/A converter output) known as the latency. The larger the block length is, the greater the processing time and thus the greater the latency of the system will be. The maximum latency for a playable musical instrument is around 10ms. Longer block sizes give a larger leeway for processing operations because there are far less function calls which waste clock cycles and prevent pipelined instructions, though many algorithms run in quadratic time meaning smaller data sets are advantageous. There is often a tradeoff in block length between processing time and latency.

Ackenhusen [17] describes an incremental STDFFT algorithm for real time DSP systems. This allows an approximate result to be output at the deadline regardless of whether the algorithm has reached its end point. The relationship between cost (time) and quality of output depends on the design of the algorithm. Algorithms such as these are considerably more difficult to develop than standard algorithms, and some algorithms do not lend themselves easily to incremental behavior.

4.2 Digital Signal Processors and Microprocessors

“A digital signal processor’s data format determines its ability to handle signals of differing precision, dynamic range, and signal-to-noise ratios. Because floating-point DSP math

reduces the need for scaling and probability of overflow, using a floating-point DSP can ease algorithm and software development.” [42]

Floating point data formats have a drastically larger dynamic range than fixed point systems since they can represent much larger and much smaller values, and much smaller quantization noise levels since the distance between each consecutive value is potentially much smaller. DSPs often have different architectures from standard microprocessors. For example the Analog Devices Super Harvard architecture (SHARC) is an optimized architecture which has three busses, two for data (40 bit) and one for a program instruction (48 bit) which allow four 32bit data transfers every core processor cycle.

Multiple processing elements containing ALU's, full parallel multipliers and shifters allow several parallel instructions to be executed on every clock cycle. DSP's often feature extended precision ALU registers to prevent build up of round off noise during accumulation of floating point values. Pipelining is a further improvement to processing time that can be made in certain situations. If the architecture allows it, this technique allows the overlap of execution in consecutive stages of each instruction which can drastically reduce computation time. Many of these improvements only increase the speed of execution for loop based algorithms, but do not make much difference for highly branched code. These architectural improvements to the computation units are combined with dual ported internal SRAM memory, which increases speed by a large amount since no external bus transactions or logic propagation delays are incurred, and allows for two simultaneous read or write operations within the memory.

Circular buffers are very useful concepts within signal processing, and some DSPs have Data Address Generators (DAGs) which provide this functionality with zero instruction overhead. They can also provide zero overhead looping and bit reversing. These instructions are typically part of an extended instruction set which uses all of the optimized hardware additions to provide assembly instructions such as multiply-add-shift. DSPs also often feature on board peripherals such as Direct Memory Access (DMA) controllers, serial and parallel ports, A/D and D/A converters, clock generators, universal asynchronous receiver transmitters (UARTs), pulse width modulation (PWM) generators etc. These mean that a minimal amount of additional circuitry is required, important since DSPs are often part of small size or embedded hardware designs.

4.3 Platform choice

It is hard to compare DSP chips across the various lines of different manufacturers, since there are many chips at varying specification in each product range, of which each manufacturer has several. The important factors when choosing a DSP to base a design upon are the following. Once the data format (either floating or fixed point, at x number of bits) has been chosen, processing power is a large consideration. This consists of more than just the clock speed however; as the architecture of computation units and extended instruction set also determine the amount of operations that can execute per second. The amount of onboard RAM is also a consideration because if a lot external transactions are required during computation then the performance will be considerably reduced. The onboard peripherals available in the DSP are also important in order to reduce external circuitry. The final consideration is the availability of documentation and library code, as well as the development kit and debugging environment and possibly a development board.

I will compare two DSP chips here, one from the aforementioned SHARC product line, and one from Texas Instruments C6000 floating point DSP family, both costing around \$20 and both based upon the Super Harvard architecture. The SHARC 21261 DSP is based upon a 150 MHz core and has 1 Mbit dual ported SRAM and 3Mbit internal ROM. The TMS320C6722B-200 has a 200 MHz clock speed, 256Kbyte internal RAM (8 Mbit) and 384Kbyte ROM (12 Mbit). The 21261 has four serial ports supporting I²S and TDM coding, one serial peripheral interface (SPI) port, and it also has two clock generators and three timers. The C6722 has three audio serial ports supporting I²S and TDM, two I²C inter processor serial ports, two SPI ports, and one internal timer. Both chips have external memory ports, advanced data movement and signal routing systems based upon DMA, as well as hardware implementation of circular buffering and complex addressing operations.

Analog Devices provide the VDK real time kernel for use with SHARC processors which provides real-time tools such as context switching, multi threading, semaphores, and scheduling. This is integrated directly with VisualDSP++ which is an integrated development environment (IDE) specifically designed for software development and debugging on Analog Devices hardware. Analog Devices also provide software modules including surround sound processing and common codecs for compressed audio formats such as MPEG-3 and WMA. There is also a large bank of example code and DSP algorithms, with and without hardware optimizations. Texas Instruments provide DSP/BIOS as a real-time operating system

(RTOS) for the C6000 platform which provides multitasking, networking, multiprocessing and hardware driver modules. It integrates with the Code Composer Studio IDE which features graphical analysis and debug tools. There is a range of library tools and common codecs.

I have decided to base my design on the Analog Devices SHARC 21261 device, primarily because of the availability of development tools, example schematics and code, application notes, and software libraries. There is little to choose between the chips in terms of performance, and a sampler does not require a great deal of complex interfacing. The interfaces required are primarily external memory, synchronous serial ports for an audio codec, a MIDI input port, and an LCD display, as well as several general purpose I/O pins for the user interface (buttons). Figure 8 shows the internal architecture of the 21261 DSP.

4.4 Analysis of Memory and Processing requirements

The FFT algorithm takes roughly $0.5 n \log_2 n$ multiply operations and $n \log_2 n$ add operations. These values are calculated for a radix-2 decimation-in-time (DIT) algorithm. There are far more optimized routines available if details of the data set are available. For an oversampling ratio of 4 (75% overlap of frames) and a block size of 1024 samples, with the forward and inverse transforms there are 122880 operations. For the ADSP 21261 at 600MFLOPS this corresponds to 0.0002048s, not taking into account any optimization of the FFT algorithms or use of extended instructions such as shift-multiply-add. The datasheet for this chip specifies a 1024 point radix-4 FFT time of $0.46\mu\text{s}$ which after reversing and oversampling is $3.68\mu\text{s}$, an improvement by a factor of 55 over the radix-2 on a non optimized architecture. This value is per key zone, and does not take into account re-sampling or FX operations, but sampling at 44100Hz allows 0.023 seconds processing per 1024 sample output block, which is a factor of 1500 larger than the time required. This value does not take into account the maximum latency figure of 10ms. These calculations also do not take into account the overhead taken by I/O functions or time taken calculating function offsets from vtables if virtual functions are used.

In terms of memory requirements, a one second waveform at 44100Hz 32 bit needs 1.411Mbits of RAM. Considering the system will be limited in terms of memory, waveforms should be stored as 16 bit signed integer values and converted to 32 bit floating point values for processing. This would half the required storage space, at the cost of a single conversion operation every time a sample is loaded from memory. A block of 1024 samples at 32 bit

takes 32768 bits. This value is much lower than the internal RAM size of the 21261 (0.5Mbit) which means that internal block operations such as transforms and effects do not need to make external bus or DMA transactions.

5. Hardware Design

5.1 Schematic Design

The ADSP21261 is a 144 pin LQFP (low profile quad flat pack) surface mount IC, measuring 20mm x 20mm with 0.5mm lead pitch. This means that the PCB needs to be manufactured to a small scale, and consideration will have to be given to the layout in order to reduce effects of high speed digital noise and cross talk, especially where the analogue and digital sections of the circuit meet around the audio codec. Figure 9 is a block diagram of the proposed hardware solution. The design began with an extensive review of literature from Analog Devices [42] [43] [44]. The Analog Devices 74111 mono audio codec was chosen because it is small, cheap and provides 24 bit sigma delta A/D and D/A conversion and gain scaling at variable sample rates with a minimum of additional circuitry. This can transmit and receive information and control bytes with the DSP via a synchronous serial port clocked at twice the selected sampling rate.

A serial flash ROM was chosen as it is small and easily interfaced using the SPI mode 3 interface. This is a full duplex, four wire interface between a master device and several slave devices. The 21261 can boot straight from this device by connecting the BTCFG flag pins correctly. These pins also select the DSP core clock speed. The specific device chosen was the ST Microelectronics M25P20 which has 2 Mbit of storage. In order to provide In Circuit Serial Programming (ICSP), a connector to each of the pins of the device was added. The power supply to the ROM chip during programming is via a diode in order to prevent powering the whole circuit whilst programming. The M25P20 is compatible with most universal programming devices.

External Static Random Access Memory (SRAM) was chosen because it is much simpler to interface than DRAM. This is because DRAM stores information on capacitors which require periodic updates. This causes additional complication for the processor, although provides faster and smaller memory. The processor has a 16 pin parallel interface, which is time multiplexed to provide either 8 bit or 16 bit access. In 16 bit mode the address can only be 16

bits wide limiting the memory to $2^{16} * 16$ bytes or 1Mbit. In 8 bit mode up to 24 bits of addressing can be used (16 bits in the first clock cycle, 8 in the second). This provides access to up to $2^{24} * 8$ (16Mbit) of memory. I have chosen to use a 4Mbit chip, interfaced with two 8 bit registers. This is a very similar set up to the ADSP21262 development board.

For user interfacing an LCD screen and 6 buttons were chosen. The ADSP21261 has 20 configurable input/output pins in a structure called the DAI (Digital Audio Interface). These can be patched in software using the signal routing unit (SRU) to any of the on chip peripherals such as serial ports, or used as general purpose inputs and outputs. The majority of available LCD displays have either 4 or 8 bits wide parallel interface. The processor has synchronous serial outputs so I used a shunt register in order to convert between the two. This incurs a delay of the clock speed multiplied by the width of the data format. However because the processor handles serial I/O with DMA controllers and LCD output is clocked by a second processor output pin, this is not a problem. The buttons are interfaced via Schmitt triggers in order to remove bounce noise, and then into a 8 to 3 binary encoder before being read by three of the DAI pins.

As described in the literature review section, MIDI is an asynchronous serial protocol. EE191 [45] describes how oversampling of the serial input can provide a method of simulating a UART receiver. This basically involves oversampling the MIDI input (which is at 31784 bps) by three times and analyzing it in software for rising or falling edges. Both the Receive Frame sync and Data in pins of a serial port must be connected to the transmit pin of the UART (in this case the MIDI in port) in order for the serial port device to synchronize to the signal. MIDI protocol requires an opto-isolator on the input in order to isolate connected devices. This is because many MIDI devices allow daisy chain connections which can cause signal loss and protects devices from other faulty equipment.

Note that Figure 9 includes an RS232 transceiver and a DAC, which are replaced in the final design by the 74111 audio codec. The serial port was intended to be used to load samples into the sampler from a computer, and also potentially to aid debugging by using bootstrap code, similar to the method used by the early SHARC EZ-Kits such as the one based upon the ADSP-21060. The modern development kits use a USB interface, but the schematics for these is not release with the designs for the rest of the board as it is proprietary to Analog Devices. Instead of the additional complexity of the serial port, the codec offers A/D conversion for input of an audio signal. Most conventional hardware samplers offer two or

three options for this: analogue audio input, SCSI data transfer, or digital audio input via SPDIF (Sony-Phillips Digital Interface Format) protocol. The latter two options are now dated and have fallen out of mainstream use, being replaced with compact flash or other solid state drives, IDE controllers, or USB or Firewire ports.

I generally aimed to keep the circuit modular so that if one section fails then it should not affect the whole board. This is made easy with the SHARC architecture since there are no shared busses and the internal processor busses are not accessible (The SPI port and parallel ports can be used in a shared way if necessary). This alleviates any bus contention issues and reduces the amount of timing issues to be considered. A header socket was connected to the CPU JTAG (Joint Test Action Group) pins for flexibility. This interface provides CPU test functionality such as boundary scan path, and also allows a programmer to access an on chip debug module and use an in circuit emulator (ICE) to aid development of code. Programmers for SHARC are typically very expensive, although there are open source programmers and software for other microprocessor and DSP ranges such as Analog Devices BlackFin and the ARM ARM7 device family.

5.2 PCB Design

National Instruments Multisim was used for the schematic design. I began by creating symbols for the processor and other devices. Multisim allows direct export and synchronization with National Instruments Ultiboard, which was used to design the PCB. The ADSP21261 datasheet strongly advises using a power plane for the VDDExt and VDDInt power supplies, as well as a ground plane. Because I required two signal planes, I could only have two power planes, since a 6 layer board was too expensive. To get around this, a large planar copper area connected to VDDInt was placed underneath the DSP chip to provide a pseudo power plane. The top surface doubles as the 5v power plane, and the bottom surface is also largely grounded. This also ensures that the PCB copper density is roughly even across all layers in both axes, preventing warping during manufacture.

Another problem was that Analog Devices instruct the use of microvias underneath the DSP pins to the power and ground planes to minimize the inductance of copper traces and vias. Unfortunately the PCB manufacturing facility can not easily produce these, so I had to resort to using regular vias with the shortest possible traces. 0603 spec capacitors are realistically the smallest I could assemble by hand on the PCB, so these were squeezed around the DSP wherever possible to decouple the V_{DDExt} . Twelve capacitors were used on the underside of

the PCB to decouple the V_{DDInt} supply. Bulk capacitors were used on V_{DDExt} , V_{DD} , A_{VDD} to provide stability in the power supply. 10uF capacitors were placed around the board on the V_{DDExt} plane, and smaller capacitors were used to decouple the power pins of all semiconductors on the board. Traces were kept as short and as wide as possible. For the memory, bus skew is a possible issue as some parallel pins are further than others from the registers. This is inevitable though, and the traces are reasonably wide since the register and memory chips are large, which counteracts the distance. Auto routing tools were not used during the design. I also tried to orientate all of the semiconductor devices in the same direction to reduce confusion during assembly.

Figures 10, 11, 12, and 13 show the schematics for the sampler. Figure 14 shows the finished PCB layout (all layers superimposed) and Figure 15 shows a 3D impression of the assembled PCB.

5.3 Assembly and manufacture of hardware

After exporting the PCB as gerber files, it was manufactured by an external company called PCBTrain. Assembly was performed by hand, using a standard temperature controlled soldering iron. A cost sheet of all components can be found in appendix 3. A link was missing in the ICSP interface which was easily fixed with a short wire run. The width of the SRAM chip footprint turned out to be slightly narrower than the PCB pads, which meant the TOJ pins, which are folded under the chip had to be bent outward in order to solder to the board.

5.4 Testing of hardware

I programmed and verified the flash chip through the ICSP interface in order to test it, using a GALEP-4 programmer. This completed successfully. On my first attempt to power up the main circuit I applied 5v to the power connector and observed the outputs of the voltage regulators. During the design of the PCB I had been looking at the data sheet for the core voltage regulator upside down. This caused the full 5v supply to run into the DSP core supply, rated at only 1.1v. The DSP was very hot suggesting internal damage. From this point, I removed the voltage regulator and applied a voltage directly from a bench supply; however I could not get the DSP to respond in any way. At the very least it should have clocked the crystal pins or tried to access the SPI interface on booting. On an oscilloscope however there were no periodic outputs.

As a result of these developments, I decided to change the focus of the project from hardware design to software. This occurred around February 2008, and I decided that there was not enough time or budget available to alter the design, get another PCB produced and assembled and then to do the software development. An alternative would have been to develop peripherals to interface with the Analog Devices EZ-kit development board. However as stated only a very old specification kit was available which would severely limit the algorithms that could be developed.

6. Chapter – Software Design

In order to get around the fact that there would be no hardware to run the software on, I decided to implement the sampler code as a VST plug-in. This would enable development and execution on a desktop computer. I developed the code using C++ in Microsoft Visual Studio 2005, using the freeware program VSTHost for testing.

6.1 Steinberg VST Implementation

A VST plug-in is a DLL file with an exported function which can be used within a host application. This can be defined in the user program or by including the file VSTPluginMain.cpp from the VST SDK. The SDK provides the class AudioEffectX as a basis for any audio effect or instrument. This is a subclass of AudioEffect which is the VST version 1.0 equivalent. An AudioEffect can take input and provide output of a stream of audio and MIDI instructions, configurable by the VstInt32 canDo(char * text) method. This allows the host program to send strings corresponding to functionality to the plug in (of any VST version) and find out whether this is supported. In this way the plug-in can act either as an audio effect or an instrument. For the sampler, this is configured to take in MIDI data and output audio.

The method VstInt32 processEvents(VstEvents *events) is used by the host to send MIDI information to the plug-in. Standard MIDI byte information as discussed in the literature review is used for this. The method void processReplacing (float** inputs, float** outputs, VstInt32 sampleFrames) is used by the host to request audio data from the plug-in. If there are no current notes being played then the output array is filled with zeros. In my code, the host is constantly requesting samples from the key zones of each currently selected patch, which normally return zero. When a MIDI note on message is received each key zone of the

patch is informed and consequently if they are set to a waveform then when requested they return the next sample value from this, after re-sampling and / or effect processing.

I have implemented two methods of creating patches and editing parameters. Firstly I have created a VST parameter based editor. This is a feature provided by the VST SDK which allows simple editing of parameters using sliders. The number of editable parameters is registered with the AudioEffectX super class on construction. For each parameter from 0 to n-1 there are five methods which are called by the host:

```
void getParameterLabel (VstInt32 index, char* label)
```

```
void getParameterDisplay (VstInt32 index, char* text)
```

```
void setParameter (VstInt32 index, float value)
```

```
void getParameterName (VstInt32 index, char* label)
```

```
float getParameter (VstInt32 index)
```

These combined allow reasonably flexible editing to be set up. The SDK example program VstXSynth uses an enum to list the parameters, and a switch statement on the index argument of each method. This means that to add an additional parameter, edits to five methods are made. The approach I have taken is to create an abstract base class UserParam with the above five methods, and int ID field. For each parameter used there is an implementation of UserParam contained in a hash map, with the key value being the ID. This allows the above five methods to simply lookup a UserParam from the index provided and call its corresponding function.

Because there are a lot of nested fields within the sampler (each keyzone has many parameters for example), rather than have a hundred or so sliders on the interface, I have implemented a hierarchal system where each UserParam may have several child UserParam pointers. This means that there can be a key zone parameter, for example, which selects which individual key zone's parameters are edited by the visible sliders. Thus only one set of sliders for key zone parameters is required. In order to do this the VST function void setParameterAutomated(VstInt32 id, float value) was used on all the children of a UserParam when it is edited in setParameter. This reduces the overall number of parameters to 28. Rather than use a slider for the current patch, I have chosen to use the VST host to select the

current patch, since this is a feature provided by the void setProgram (VstInt32 program) method.

The second GUI implementation is an emulation of the hardware sampler interface. This uses 6 buttons (up, down, left, right, plus and minus) and a 16 x 2 LCD screen to display and edit all parameters in a hierachal menu system. This uses the CFrame class from the VSTGUI toolkit which is a separate SDK by Steinberg which provides platform independent GUI widgets. AEffGUIEditor is the class which is extended in order to provide this functionality. MenuManager is a class which contains the storage of MenuItem objects in a nested heirachy. Each MenuItem is a node in the menu tree with four connections and two pointers which point to member functions of a created FunctionManager object. This design follows the Model View Controller (MVC) design pattern. The Sampler object is the model, the MenuManager provides the view, and FunctionManager is the controller. This allows easy editing of the visual appearance or underlying functions without affecting the other. This may be useful if the Sampler and other audio code is to be used and altered for a different platform. The design of the software is fully object oriented, with the only standalone function being the VST function called by the DLL: AudioEffect* createEffectInstance (audioMasterCallback audioMaster).

6.2 FX Code

DelayFX is a class implementing a simple delay effect with adjustable feedback, timing, and mix. The input of the effect is sent to the output where it is mixed with the sample on the end of the buffer according to the mix parameter. The input sample plus the delayed sample multiplied by the feedback ratio are pushed onto the front of the buffer. The timing parameter determines the length of the delay line. This effect uses very little processing power and extends the TimeDomainFX class.

ThreeBandEQ is a class implementing a simple 3 band non parametric EQ for simple modification to the sound. This class uses multiple inheritance to reduce the processing power required. It inherits the FrequencyDomainFX class which instead of the double f(double x) method in TimeDomainFX has the double* f(double*, int block_size) method. This allows an object to process frequency domain information. Because 3BandEQFX inherits both TimeDomainFX and FrequencyDomainFX it can be used either in the KeyZone FrequencyDomainFX field, or in the Patch TimeDomainFX field. Multiple inheritance can be a problem when a class extends two derived classes of the same base class. As both

3BandEQFX's parent classes are sub-types of the class FX, this situation is present, but since neither TimeDomainFX nor FrequencyDomainFX override the pure virtual functions in FX, there are no problems in resolving the function table. This object oriented approach to FX means that more code is used, but there is an overall reduction in the computation in certain situations. It provides a framework for FX classes to be developed with a "pluggable" feel.

Within a patch the TimeDomainFX field defaults to a static object

TimeDomainFX::NULL_FX which simply passes the audio samples straight through with no effect. A different TimeDomainFX object may be assigned to this field and thus effect the signal. The FrequencyDomainFX class acts in the same way within the KZone class.

6.3 Custom Hardware Development

Software development for the custom hardware would have been considerably slower than on a PC. This is because on my hardware there is no debug environment due to cost restrictions. The core I/O functions would have been developed first, involving setting up the LCD display and button interfacing. This would allow information to be passed for debugging. Next the external memory would be set up and functions written allowing allocation and access of this using the DMA controllers with similar methods to the C standard library functions void* malloc(int) and void free(void*). The next step would be to set up two way communication with the audio codec for playback and recording, and basic wavetable output established. Finally MIDI communication on a serial port would then need to be established.

From this point it should be possible to include the DSP functions such as re-sampling, effects, and filtering one by one. Much of this code could be tested and developed on a PC before transferring onto the hardware, making it much easier to step through code and debug it.

7. Results

Figures 16 and 17 are photos of the assembled hardware project. Figure 18 is a class diagram for the software project. Figures 19 and 20 are screenshots from the VST plug-in. The tests devised in the conclusions of the literature review section were designed to test the hardware device and thus were not entirely applicable to the VST implementation. I have followed the tests and adapted them where necessary and present them in the following section.

It is hard to devise a measure of output quality for re-synthesized audio, because by nature it is deformed from its input function. Sampling synthesis is often looked at as a way of producing realistic sounding synthesized instruments. Massie [3] discusses the two key parameters of musical synthesis, accuracy and expressivity of sound. He also discusses these as a function of cost, in terms of memory and processing power. It is also important to consider the types of input that may be used. For example, there is little practical use in considering the output of the pitch shift algorithm on a waveform of a drum beat, as there is no pitch to detect. The key algorithm, re-sampling, uses the linear interpolation method and given an input of a musical note successfully produces an output at a pitch transposed up by the specified amount of semitones according to equal temperament tuning. It is possible to hear imaging distortion when interpolating the waveform since no anti-imaging filtering takes place. I have implemented the following effects: delay, pitch shift, three band eq, and one band parametric eq. All of these function correctly and have adjustable parameters. Although the sampler has four voices, it is monophonic as only one note can be produced at once. Polyphonic operation was something that was in the project aim, but it was not achieved. Each of the four voices have amplitude and filter envelopes, volumes, key ranges, and sustain looping.

The cost of the VST plug in is high in terms of the processor usage. It is possible to stop real-time output of the sampler by trying to use too many concurrent waveforms or effect algorithms. By this I mean that the blocks of audio can not be processed in time to output to the host and the output is interlaced with blocks of silence. This is obviously unacceptable for a musical application. The following CPU usage figures are from tests run on a 1.8Ghz AMD Laptop in Visual Studio Debug mode using VSTHost with several other applications loaded. No patch or key zone effects were used, although filters were still evaluated. All waveforms used were identical 2 second clips of a violin at 44100Hz. The baseline CPU usage was around 43%, peaking around 52% for one waveform loaded. When two waveforms were loaded the base CPU usage was 61% and the peak when played was 70%. When three waveforms were loaded, the usage is 72%, going up to 80% when played. When four waveforms were loaded, it caused the base CPU usage to be 85% and the sampler can not manage real time output.

When re-sampling a violin, the quality of the output is acceptable. There is a noticeable distortion or glitch where the audio blocks are spliced back together (every $1024/44100$ s),

although this is not significant enough to stop the instrument being recognizable. The key zone filter is surprisingly “musical” considering it is a simple two line frequency response model. The patch filter, delay effect and eq effects are also quite effective in creating interesting sounds. When playing back a drum loop there appears to be some distortion, possibly caused by clipping or quantization between float and double precision values. The pitch shift effect does asynchronously re-pitch relatively effectively, although a slightly metallic sounding distortion is introduced. This is likely due to harmonics being incorrectly shifted.

Both of the user interfaces were tested thoroughly in order to check there were no bugs in the parameter editing code. The VST parameter interface successfully loaded samples, edited all parameters and effects. The VST GUI interface successfully allowed the same level of manipulation as if it were a 16 x 2 LCD display with 6 buttons. There are a lot of editable parameters which means that the menu system is quite complex and it is possible to get slightly disorientated at first. Both interfaces successfully kept each other updated.

8 Discussion

The Gantt chart in Figure 1, was not particularly accurate in its estimations of duration of each stage. Design of the schematic and PCB took vastly longer than expected. The PCB design was finished and checked around January 10th 2008, and manufacture of the PCB then took three weeks, and then a further three weeks to assemble. To counteract this I started working on the software on a PC when I ordered the PCB for manufacture. I worked on the main audio classes using the Simple Direct Media (SDL) library for audio functions and user I/O. Much of this code is used in the VST implementation. In hindsight I should have started this before I started schematic or PCB design, as development of the software led me to think much more depth about the actual requirements of the system, and the problems that I would have to solve.

It is hard to review the PCB and hardware design beyond the aforementioned power regulator and ICSP faults, because testing never got beyond the power up stage. If I were to design the PCB again from scratch I would use all surface mount components, as opposed to a mixture of these and through holes components. After experiencing the difficulties of developing signal processing algorithms, I would also be hesitant to design another board without a debugging interface. The expense of SHARC JTAG programmers makes this a prohibitive

option, but there are open source equivalents for other capable chips. It would have been very beneficial to develop software on the SHARC development board; however the only one available to me was based around a much older DSP and with no external memory, which limits its usefulness for a waveform sampler. I found that device placement and orientation are very important factors in minimizing routing complexity on the PCB, and that time spent at this stage saves a lot of time later spent routing tracks.

For the software design, I think that the code is relatively well structured in terms of object orientation. The class structures represent real world “entities” and implement the four fundamentals of OOP, inheritance, encapsulation, abstraction, and polymorphism in order to make the code much more understandable and flexible. The audio quality is slightly lower than desired. The noticeable distortion from block processing could be improved by appropriate use of oversampling. This would also improve the analysis step and thus the output quality of the pitch shift effect. The performance of the sampler in terms of processor usage is not good, and improvements to this must be made before oversampling can be implemented. A profiler program should be used to determine the points in the code where most clock cycles are being used and appropriate optimizations can be implemented for these sections. The 80/20 rule should be used here, in that generally 80 % of execution time is spent in 20 % of the code, and these are the sections which should be improved.

For the software the development model I followed was similar to the Agile process. Agile is a modern methodology which promotes iterative development throughout the life cycle of the process, as opposed to a highly linear traditional model of analysis, planning, design, coding and testing. Agile is beneficial because it means that the requirements and structure of the project can change without large amounts of wasted time which you would get with the waterfall process. In the case of this project, the aims of the project changed drastically at which point a lot of the planning and analysis which I had done became redundant.

If I were to begin the project again then it would be planned in a very different manner. The first thing I would tackle would be software implementation of audio algorithms, initially for PC and then on the SHARC EZ-Kit development board. This stage really improved my understanding of the architecture of the system required, the processing and memory requirements, and also the time scale of the project. When I had a code base to work from and a thorough understanding of the peripheral control and SHARC I/O operations and DMA control, I would begin rapid design and prototyping of the hardware, followed by low level

software development for the hardware. In this sense I feel that by developing the hardware first I approached the problem backwards, even though this was the path recommended by many texts. This is probably due to their assumption that the developer will be experienced with DSP coding before commencing the project.

The aim specifies 16 bit 44.1 kHz sampling operation. The VST plug-in currently runs at 44.1 kHz, but it could be modified re-sample waveforms to the given rate when they are loaded, or to scale the sample-rate conversion ratio of each waveform dependant on its recorded sample rate in relation to both the note played and the output sample rate. The VST does not run at a given bit rate, it depends only on the file being opened. The library used to import waveforms, MiniAiff, opens AIFF audio files of any bit rate. The internals of the code use single and double precision floating point variables. I aimed to use double fields for intermediate calculation steps and float fields to store values. This is a similar approach to the double precision accumulator used in the SHARC processing units. The noise on the audio outputs could be reduced by using double values throughout the code, although this would increase computation time and memory usage. Single precision floating point values are perfectly capable of providing effective noise free operation however, when compared to the practical noise floor present in any recorded waveform or noise floor of AIFF or WAV files which are 16 or 24 bit signed integer formats.

8.1 Developments from the project

The most obvious development to the sampler would be to add polyphonic synthesis. Allowing several notes to be triggered simultaneously would allow chords to be played, adding to the realism of synthesis, because for example a piano is rarely played with only one finger. Polyphony does not necessarily require a great deal more computation than monophonic operation. Typically the attack portion of an envelope for a key zone begins when a note is pressed and the release portion begins when the key is released. When multiple notes are triggered, the attack portion is triggered by the first note, but subsequent notes do not trigger this, the release portion coming when no notes are left held down. As a result of this, the stored waveform can be re-sampled to the specified pitch for each note currently held down, and then these values can be added together before the key zone envelope and effect processing, these being the two components which use most processing power. To implement this a table storing the current status of each MIDI note would be required. To limit the amount of simultaneous notes allowed (in order to maintain real-time performance), a record of the relative order of notes triggered would also need to be stored so

that the first note triggered is the first to be switched off if the limit is exceeded. It is worth noting that there are a large number of monophonic synthesizers, for example the Yamaha CS20 and Dave Smith Instruments Evolver, and it is not as restrictive to creative sound design as may be imagined, although nearly all sample based synthesizers are multi-timbral and polyphonic since the technological advancement is small.

Low Frequency Oscillators (LFOs) with patchable modulation would allow a wider variety of synthesis options. They could be patched to the pitch shift, filtering, or effect parameters to allow sounds which develop over time. Reverb could be added as an effect. As stated in the literature review section, this is a very widely used and versatile effect. It could be implemented as either a FrequencyDomainFX by using the convolution reverb method, or as a TimeDomainFX by using a combination of all-pass filters and delay lines to simulate the early and late reflections of a room or space. There are also several further common effects which could be implemented, including chorus, flanger, phaser, tremolo and compression. These are combinations of delay and time based modulation, except the latter effect which is used to reduce the dynamic range of its input. Nearly all synthesizers have pitch bend controllers, which allow instantaneous pitch shift of the notes being played (in the case of sampler, the re-sampling ratio) up or down by a real value between plus and minus two semitones. This is supported in the MIDI protocol. Another addition could be wavetable mixing, as seen on the Korg Wavestation, involves delaying the starts of, and cross fading between key zones to produce slowly evolving tones. Formant analysis and correction would be a useful feature to add for synthesis of vocal samples. This would allow greater and more realistic shifts of pitch to be performed.

As we are currently reaching the limit of CPU processing power, we should consider the possible optimizations which can be made to the speed of the algorithms. Firstly functions which are currently calculated in real time, for example window functions, can be stored in lookup tables. This is a trade off between processing time and memory space. Since the desktop PC has a much larger standard memory than the hardware design, this is a sensible choice. Use of deadline driven programming and scheduling would allow the sampler to calculate its real-time performance and make decisions based on this information at run time on the quality or type of algorithms used. As a VST plug in there is no way of calculating real-time feasibility checks since there are many underlying system processes including the operating system and the VST host. However CPU usage could be monitored and simpler

algorithms, shorter transform lengths and fewer coefficients could be used if high levels are detected. The VST plug in does not currently implement many of the optional methods such as reporting desired block lengths, input and output latency, and offline processing. The plug in could potentially provide multiple audio outputs from several MIDI inputs, thus producing multi-timbral operation.

The VST GUI simulating the LCD screen of the hardware is not a particularly suitable editor for a PC interface where mouse and keyboard input is available. A much more practical and useable GUI could be developed, either using the VST GUI toolkit, which is fairly rudimentary, or using platform dependant toolkits. It is worth noting that many VST plug-ins which are software emulations of classic hardware use an interface which simulates the real hardware, for example the Waves API Collection (see figures 21 and 22). Since VST plug-ins only run on Windows and OS/X cross platform development would not be too much more difficult. Alternatively custom controls could be written based upon the VST GUI code which would provide flexibility and cross platform support.

An advancement of FX classes would be to implement the Steinberg VST plug-in protocol inside the sampler. This would allow the sampler to host plug-ins. This would allow a wide variety of effects to be used. VST Host programs typically allow daisy chaining of VST plug-ins however, so in the context of VST instrument this is not that valuable. If the sampler was also implemented as a standalone executable as several other VST sampling packages are (for example E-mu Emulator X) then it would be useful. On the hardware side however, allowing the device to run proprietary plug-ins would be an interesting feature. Third party developer effect development for a hardware product such as a sampler would be a major advantage in the marketplace, since there are many companies devoted to development of high quality audio FX algorithms.

However, VSTs are predominantly developed and compiled for Windows or OS X, and the Intel x86 architecture, which means that they cannot be run on SHARC hardware. A possible solution to this would be to use a multiprocessor based system, with an x86 compatible processor supplementing the DSP running an embedded version of Linux and Wine (an open source Windows emulation layer). This processor could also be used to perform all of the interfacing with peripherals (perhaps including more complex ones such as an IDE controller for a hard-drive and/or optical drive for storage of patches), or dynamic RAM. This would mean the DSP could be used solely for audio algorithms.

8.2 Future of DSP hardware in audio synthesis

This is an approach similar to that taken by the Korg Oasys, which is powered by a 2.8Ghz Pentium 4 processor running Linux. Although more than capable in terms of processing power, it does not allow VST plug-ins. This is mainly for marketing reasons, and also so that the developers can maintain reliability, as introducing untested third party code is always a risk. The Muse Receptor is one of the few hardware units which can currently use VST plug-ins, based on the AMD Athlon 2500+ processor. The retail price of £1400 means that it is more expensive than high spec personal computers capable of running an equal if not greater amount of plug-ins by using standalone VST host software such as XLutop, EnergyXT etc. SM Pro Audio have recently announced the release of three new hardware VST hosts, aimed at studio musicians, guitarists and DJs / live musicians respectively [46]. Whilst VST as an open architecture is not widely supported on dedicated hardware, there are a number of other DSP based audio processing systems for PC, each of which have proprietary formats, for example the TC Electronic PowerCore, the Universal Audio UAD system, and the DigiDesign RTAS format for ProTools. The latter option can be run either on dedicated DSP PCI cards or on the host CPU, whereas the former two options run solely on dedicated DSP hardware via Firewire or the PCI bus.

There is currently a surge of digital music hardware which is flexible in its application. The Roland VariOS discussed earlier was a very powerful and flexible system, but was limited in its success because its flexibility relied upon proprietary updates. The Korg Oasys (an acronym for Open Architecture Synthesis Studio) follows a similar path, but since it is the company's flagship product it receives a lot of updates and attention, serving as a test bed for synthesis and effect algorithms, and many of these filter down to their other products over time. The Alesis Fusion is a much cheaper alternative to the Oasys, providing a mix of sample based, physical modeling and FM synthesis, with multi-track audio recording. This product was recently discontinued, despite having a loyal fan base, due to low volume sales, widely agreed to be attributed to poor marketing and an initial bug ridden model which gave the product a bad name.

Line6 have recently announced the release of a new guitar effect pedal, which is essentially a DSP development kit, providing a preamp, A/D, D/A conversion, a 100MIPS Freescale DSP and programming/debugging software, allowing home users to write their own DSP code for audio manipulation. [47]

9. Conclusions

Whereas in the 1970's and 1980's the only synthesis options were provided as parameters by the manufacturer, the 1990's saw for the first time on a large scale, sampling as the new expressive and creative parameter as a way to introduce new sounds. The new millennia then saw the rise of software instruments and effects as personal computing processing power and memory increased. The next decade may see the development of optimized hardware running user customizable algorithms for truly flexible synthesis and creative expression.

Although my project was unsuccessful in terms of development of a fully working hardware product, I have explored sampling synthesis and signal processing for audio both theoretically and in commercial terms. I have learned a great deal both in the areas of managing a large project and in the technical field of digital signal processing, as well as technical writing and research skills.

I have produced a working VST sampling synthesizer which fulfils all of the bullet points in the project aims except for polyphonic operation, as explained in the discussion section. These include MIDI controlled sample rate conversion, sustain looping, envelope modulated amplitude and filters, multiple waveform mixing and basic effect implementations. There are many developments which can be made to this project both in terms of features and quality. The hardware design, though unsuccessful could be developed further into a low cost, small size sampling product which would be a useful and versatile complement to a MIDI musician's arsenal.

10. References

- [1] **Burk, P.** "*Direct Synthesis versus Wavetable Synthesis*". [Online] 2004.
http://mobileer.com/wp/direct_vs_wavetable.pdf.
- [2] **Roads, C.** "*Sampling and Additive Synthesis*" and "*Multiple Wavetable, Wave Terrain, Granular and Subtractive Synthesis*" in *The Computer Music Tutorial*. s.l. : The Massachusetts Institute of Technology press, 1996.
- [3] **Massie, D.** "Wavetable Sampling Synthesis" in . [ed.] K. Brandenburg M. Kahrs. "*Applications of DSP to Audio and Acoustics*". 1998.
- [4] **Trautmann, D.S. and Cheung, N.M.** "Wavetable Music Synthesis for Multimedia and Beyond". *Multimedia Signal Processing, IEEE First Workshop on*. 1997.

- [5] **Vail, M.** *Vintage Synthesizers: Pioneering Designers, Groundbreaking Instruments, Collecting Tips, Mutants of Technology*. s.l. : Miller Freeman Books, 2000. Information taken from <http://www.synthmuseum.com>.
- [6] **Ake, K.D and George, A.D.** The design of a microprocessor-based digital sampler. *System Theory Proceedings., Twenty-First Southeastern Symposium on*. March 26-28, 1989, pp. 349 - 352.
- [7] **Akai Professional.** S-950 MIDI Digital Sampler Usage Manual. *Introduction*. [Online] 1998. <http://oldschoolsound6.free.fr/manuels/S950Manual.pdf>.
- [8] **Hyun, M.H, Hyun, J.H and Hwang, S.Y.** Design of a pipelined music synthesizer based on the wavetable method. *Consumer Electronics, IEEE Transactions on*.
- [9] **Cho, W.J, et al.** Double-pipelined 32-voice wavetable synthesizer. *Consumer Electronics, Proceedings of International Conference on*. 7-9 June 1995, pp. 272 - 273.
- [10] **Ensoniq Corp.** *ENSONIQ ES-5506 - "OTTO"*. [Online] 1997. http://web.archive.org/web/19980214042324/www.ensoniq.com/multimedia/semi_html/html/otto.htm. (archived)
- [11] **Wyse, A.** Structure of the AWE32. *AweDomain*. [Online] <http://www.zipworld.com.au/~wyse/awedomain/intro.html>.
- [12] **Hollander, R. M.** *Turtle Beach MultiSound Project*. [Online] May 2004. <http://www.alasir.com/software/multisound/index.html>.
- [13] **Technologies, Steinberg Media.** VST Audio Plug-ins SDK. *3rd Party Developers*. [Online] http://www.steinberg.net/324_1.html.
- [14] **Roland Corporation.** *Vari-OS: Complete Real time Audio Control for Mac and PC*. [Online] <http://www.roland.com/products/en/VariOS/index.html>.
- [15] **McGill, J. F.** "An Introduction to Digital Recording and Reproduction" in "Digital Audio Engineering". [ed.] J. Strawn.
- [16] **Ling, F.P, Khuen, F.K and Radhakrishnan, D.** An audio processor card for special sound effects. *Circuits and Systems, Proceedings of the 43rd IEEE Midwest Symposium on*. 2000, pp. 730 - 733 vol.2.
- [17] **Ackehusen, J. G.** *Real-Time Signal Processing: Design and Implementation of Signal Processing Systems*. s.l. : Prentice Hall, 1999.
- [18] **Moorer, J. A.** The Lucasfilm Digital Audio Facility. *Digital Audio Engineering*. s.l. : A-R Editions, Inc, 1985.
- [19] **Grover, D. and Deller, J.** "Changing Sample Rates". *Digital Signal Processing and the Microprocessor*. s.l. : Motorola Press, 1999.

- [20] **Fitzgerald, R. and Anderson, W.** Spectral distortion in sampling rate conversion by zero-order polynomial interpolation. *IEEE Transactions on Signal Processing*. June 1992, Vol. 40, 6, pp. 1576-1579.
- [21] **Crochiere, R.E. and Rabiner, L.R.** Interpolation and Decimation of Digital Signals: A Tutorial Review. *Proc. IEEE*. March 1999, Vol. 69, 3.
- [22] **Estephan, H.S., Sawyer, D. and Wanninger, D.** Frequency Shifting. *Real-Time Speech Pitch Shifting on an FPGA*. [Online] Villanova University Department of Electrical and Computer Engineering, Spring 2006.
http://www56.homepage.villanova.edu/scott.sawyer/fpga/II_frequency_shifting.htm.
- [23] **Flanagan, J.L and Golden, R.M.** "Phase Vocoder". *Bell Systems Technology Journal*. November 1966, Vol. 45, pp. 1493--1509.
- [24] **Portnoff, M.** Implementation of the Digital Phase Vocoder Using the Fast Fourier Transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*. June 1976, Vols. ASSP-24, 3, pp. 243-248.
- [25] **Moorer, J.A.** The Use of the Phase Vocoder in Computer Music Applications. *Journal of the Audio Engineering Society*. February 1978, Vol. 26, 1/2, pp. 42-45. presented at 55th AES Convention, October 29 - November 1, 1976 in New York.
- [26] **Bernsee, S.** Pitch Shifting using the Fourier Transform. *The DSP Dimension*. [Online] September 21, 1999. <http://www.dspdimension.com/admin/pitch-shifting-using-the-ft/>.
- [27] **Puckette, M.S and Brown, J.C.** Accuracy of the Frequency Estimates Using the Phase Vocoder. *IEEE Transactions on Speech and Audio Processing*. 1998, Vol. 6, 2.
- [28] **S.S Abeysekera, K.P Padhi.** An Investigation of Window Effects on the Frequency Estimation Using the Phase Vocoder. *IEEE Transactions on Audio, Speech, and Signal Processing*. July 2006, Vol. 14, 4.
- [29] **Laroch, J. and Dolson, M.** New Phase-Vocoder Techniques for Pitch-Shifting, Harmonizing and other Exotic Effects. *Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*. Oct 17-20, 1999. Joint E-mu/Creative Technology Center.
- [30] **R.V Cox, R.E Crochiere, J.D Johnston.** Real-Time Implementation of Time Domain Harmonic Scaling of Speech for Rate Modification and Coding. *IEEE Journal of Solid-State Circuits*. February 1983, Vols. SC-18, 1.
- [31] **Bernsee, S.** Time Stretching and Pitch Shifting of Audio Signals- An Overview. *The DSP Dimension*. [Online] August 18, 1999. <http://www.dspdimension.com/admin/time-pitch-overview/>.
- [32] **Lehman, S.** Effects Explained: Delay. *Harmony Central*. [Online] <http://www.harmony-central.com/Effects/Articles/Delay/>.

- [33] **Lehman, S.** Effects Explained: Reverb. *Harmony Central*. [Online]
<http://www.harmony-central.com/Effects/Articles/Reverb/>.
- [34] **Gardener, W.** “Reverberation Algorithms” in . *Applications of Digital Signal Processing to Audio and Acoustics*. 1996.
- [35] **Wikipedia.** Musical Instrument Digital Interface. [Online]
http://en.wikipedia.org/wiki/Musical_Instrument_Digital_Interface.
- [36] **MIDI Manufacturers Association.** The Technology of MIDI. [Online]
<http://www.midi.org/about-midi/abtmidi.shtml>.
- [37] **Wikipedia.** Equal Temperament. [Online] March 10, 2008.
http://en.wikipedia.org/wiki/Equal_temperament.
- [38] **Ambler, S.W.** Introduction to Test Driven Design. [Online] December 2005.
<http://www.agiledata.org/essays/tdd.html>.
- [39] **Wikipedia.** Agile Software Development. [Online] November 29, 2007.
http://en.wikipedia.org/wiki/Agile_software_development.
- [40] **Jones, D. L.** Parks-McClellan FIR Filter Design. *Connexions*. [Online] February 25, 2007. <http://cnx.org/content/m12799/latest/>.
- [41] **Wikipedia.** Resource Interchange File Format. [Online] April 22, 2008.
http://en.wikipedia.org/wiki/Resource_Interchange_File_Format.
- [42] **Analog Devices Inc.** *Analog Devices 2126x Core Manual*. [Online] 2004.
<http://www.analog.com/en/prod/0,2877,ADSP%252D21261,00.html>.
- [43] **Analog Devices Inc.** *Analog Devices 2126x Peripheral Manual*. [Online] 2004.
<http://www.analog.com/en/prod/0,2877,ADSP%252D21261,00.html>.
- [44] **Prabhugaonkar, A.V and Comaschi, A.** EE-305: Designing and Debugging Systems with SHARC Processors. *SHARC Application Notes*. [Online] November 2006.
http://www.analog.com/UploadedFiles/Application_Notes/456074882389076171EE_305_SHARC.pdf.
- [45] **Ledger, D.** Implementing a glueless UART using the SHARC® DSP SPORTs. *Engineer to Engineer Note 191*. [Online]
http://www.analog.com/UploadedFiles/Application_Notes/399447663EE191.pdf.
- [46] **SM Pro Audio.** Free your VST's. *Press Release*. [Online] March 12, 2008.
http://www.smproaudio.com/index.php?option=com_content&task=view&id=73.
- [47] **Line6 Inc.** LINE 6 ANNOUNCES TONECORE™ DSP DEVELOPER KIT . *Latest News*. [Online] January 17, 2008. <http://uk.line6.com/news/pressReleases/644>.

11. Appendices

Appendix 1 - Test Criteria

General:

- Does sampler boot up and respond to user interaction?
- Are menus navigable and logical?
- Are parameters editable within reasonable ranges?
- Do parameters begin with reasonable default values?

Sample recording:

- Is the sampler able to record and consequently play back sound?
- Is the sampler able to save the sample in memory

Sample editing:

- Can the start and end points be edited on a sample by sample basis?
- Can the volume and name be edited adequately?
- Is this information saved correctly?

Patch creation:

- Can a new patch be created and named?
- Can a sample be assigned to the patch?
- Are samples correctly transposed to a new pitch?
- Does the amplitude envelope operate correctly?
- Does the patch filter operate correctly?
- Does the filter envelope operate correctly?
- Does the effect algorithms operate correctly?
- Do the parameters affect the sound of the effects in the correct way?
- Is the patch information stored correctly?

Playback:

- Does the sampler respond to MIDI on messages?
- Does the sampler respond to MIDI off messages?
- Does the sampler respond to patch change messages?
- Does the sample respond to other MIDI information?
- Do the key ranges for each sample work?

Appendix 2 - Menu Structure

1. Play Mode [shows currently playing patch, active MIDI channel, currently pressed MIDI note]
2. Patch Edit Mode:
 - a. Create New Patch
 - b. Patch edit
 - i. Name
 - ii. Volume
 - iii. Filter Cutoff
 - c. Sample Zone 1 edit
 - i. Select Sample
 - ii. Key pitch
 - iii. Loop start
 - iv. Loop end
 - v. Amplitude ADSR envelope edit
 1. Attack
 2. Decay
 3. Sustain
 4. Release
 - vi. Filter ADSR envelope edit
 1. Attack
 2. Decay
 3. Sustain
 4. Release
 5. Amount
 - vii. High Note (C0-C7)
 - viii. Low Note (C0-C7)
 - d. Sample Zone 2 edit
 - i. (sub menu same as c)
 - e. Sample Zone 3 edit
 - i. (sub menu same as c)
 - f. Sample Zone 4 edit
 - i. (sub menu same as c)
 - g. Patch Effect edit
 - i. Type
 - ii. Mix Amount
 - iii. Parameter 1
 - iv. Parameter 2
3. Sample edit Mode:
 - a. Start point
 - b. Trim
 - c. Volume
 - d. Name
4. (Record Mode) [record from A/D converter]
 - a. Record into buffer
 - b. Play buffer
 - c. Save buffer

Appendix 3 – Bill of Materials

Main PCB (Prices from Farnell.com for individual units as of 03/02/2008)

- 1 x 4 Layer 100mm x 100mm PCB £79.90
- 1x ADSP212161 DSP £10.55
- 1x AD74111 Audio Codec £3.03
- 1x ST Microelectronics M25P20 Flash ROM £2.75
- 1x CY7C1049CV33 4Mbit SRAM £5.42
- 1x ADM708 Reset Supervisor IC £0.73
- 1x TS1117BCP12 1.2 v voltage regulator £0.44
- 1x LD1117AV33 3.3v voltage regulator £0.66
- 2 x 74LVC573PC Octal D-Type Latch £0.77
- 1 x 74LVC148N 8 to 3 bit binary encoder £0.24
- 1 x 74LVC164N 8 bit shunt register £0.34
- 1x 25MHz crystal £0.25
- 1 x BLM18AG102SN1D Ferrite bead £0.07
- 1 x 1N3494 Diode £0.10
- 1 x 4 way DIP switch £0.96
- 1 x PCB push button £0.01
- SMT Capacitors and Resistors
- Bulk capacitors

MOLEX connectors

Interface board

- 1 x 74HC14N Hex Schmitt Trigger £0.30
- 6 x PCB button £0.01
- 5 pin DIN socket £0.53
- 6N139 Optocoupler £0.60
- 1 x 16 x 2 Parallel interface Alphanumeric LCD Module £5.62
- 2 x 3.5mm 2 pole mini jack sockets £0.35

MOLEX Connectors

Strip Board

Appendix 4 - List of files

Header Files

FunctionManager.h
MenuManager.h
LCD.h
MenuItem.h
ParamClasses.h
UserParam.h
VSTSampler.h
VSTSamplerGUI.h
FX/DelayFX.h
FX/OneBandParametricEQ.h
FX/ThreeBandEQ.h
FX/PitchShift.h
AudioClasses/ADSR.h
AudioClasses/FX.h
AudioClasses/TimeDomainFX.h
AudioClasses/FrequencyDomainFX.h
AudioClasses/KZone.h
AudioClasses/Patch.h
AudioClasses/Sampler.h
AudioClasses/Wave.h
Algorithms/FFT.h
Algorithms/IIR.h

Source Files

FunctionManager.cpp
MenuManager.cpp
LCD.cpp
UserParam.cpp
VSTSampler.cpp
VSTSamplerGUI.cpp
VSTSamplerProc.cpp
FX/DelayFX.cpp
FX/OneBandParametricEQ.cpp
FX/ThreeBandEQ.cpp
FX/PitchShift.cpp
AudioClasses/TimeDomainFX.h
AudioClasses/FrequencyDomainFX.h
AudioClasses/KZone.h
AudioClasses/Patch.h
AudioClasses/Sampler.h
Algorithms/FFT.cpp
Algorithms/IIR.cpp

Resources

Bitmap1.bmp
Bitmap2.bmp
Readme.txt

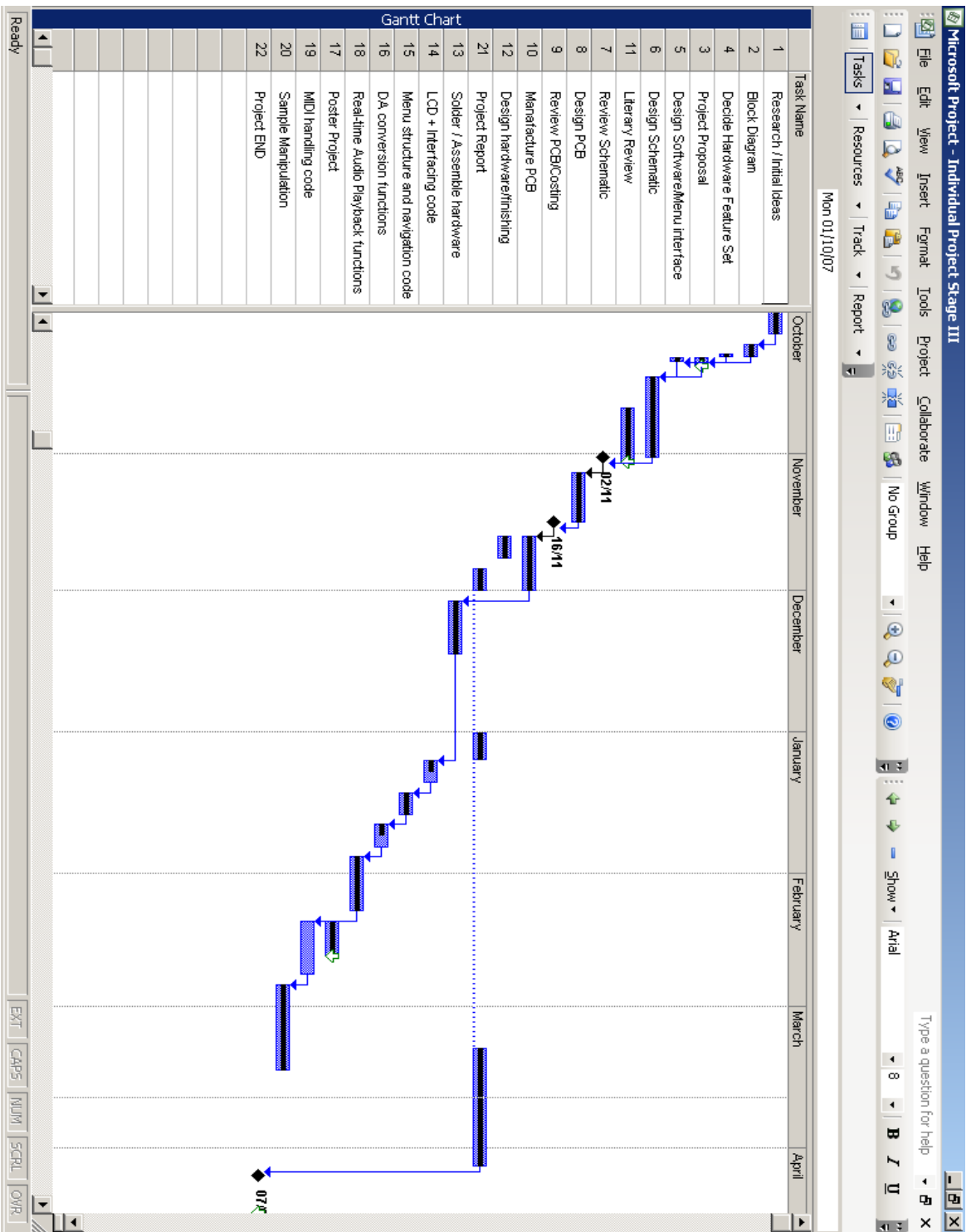


Figure 1 - Gantt Chart

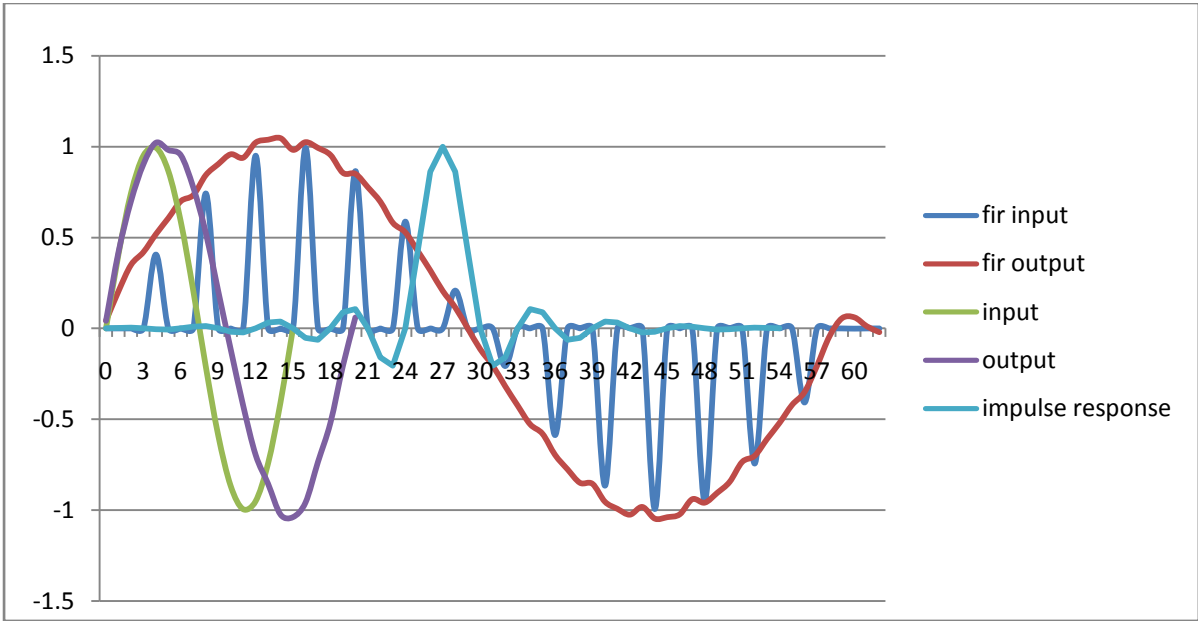


Figure 2 – Re-sampled Signal by ratio of 4/3

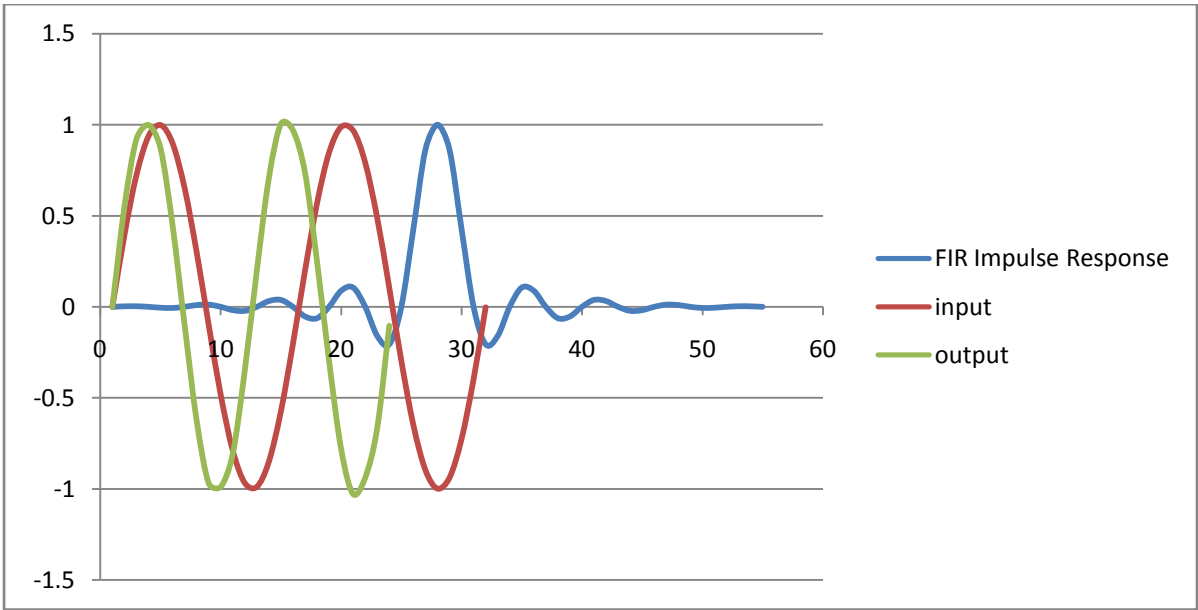


Figure 3 - Polyphase re-sampled signal by ratio of 4/3

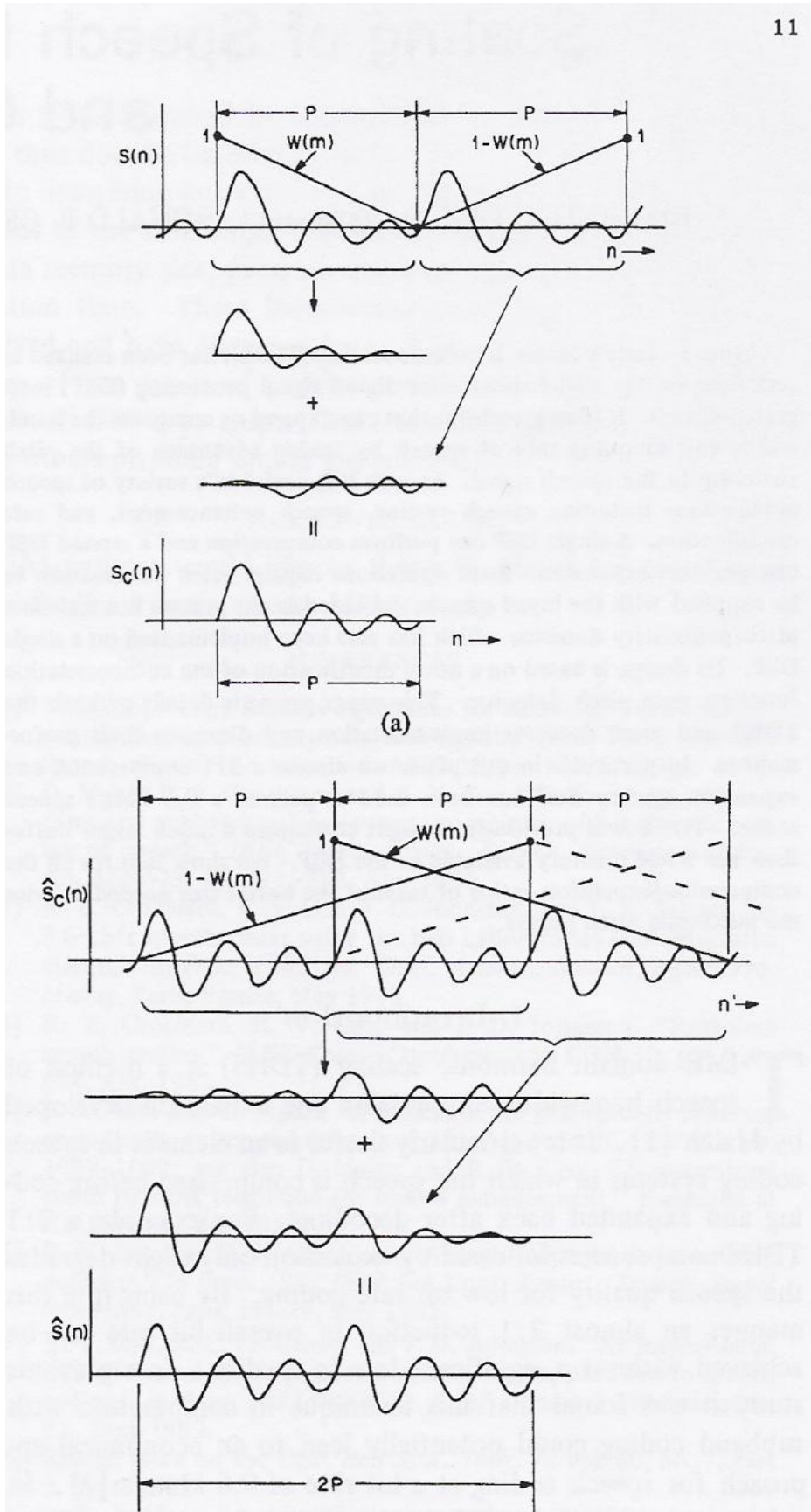


Figure 4 - Compression and Expansion of the time axis using TDHS (taken from (30))

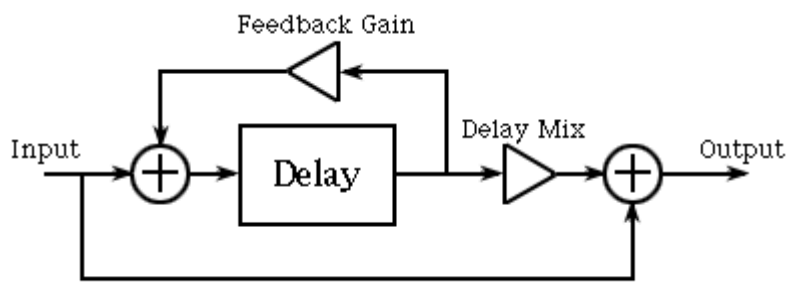


Figure 5 - Delay effect structure, taken from (32)

Impulse Response

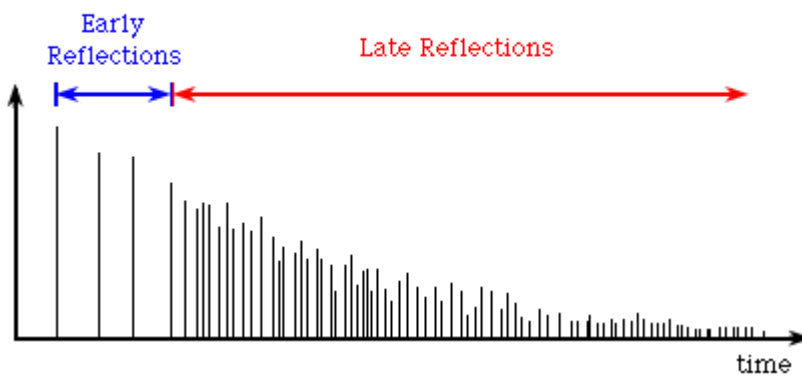


Figure 6 - Reverberation Impulse Response taken from (33)

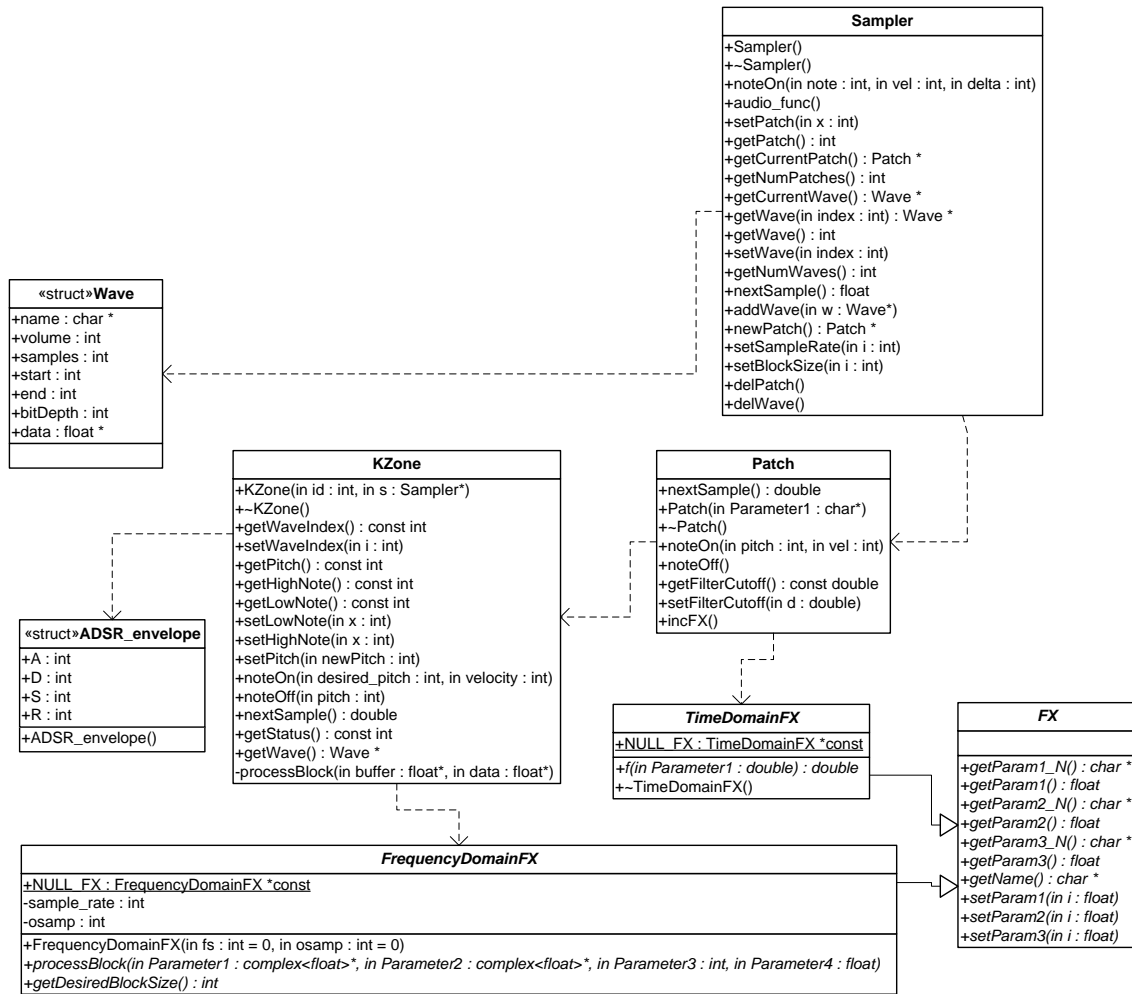


Figure 7 - Data Structures

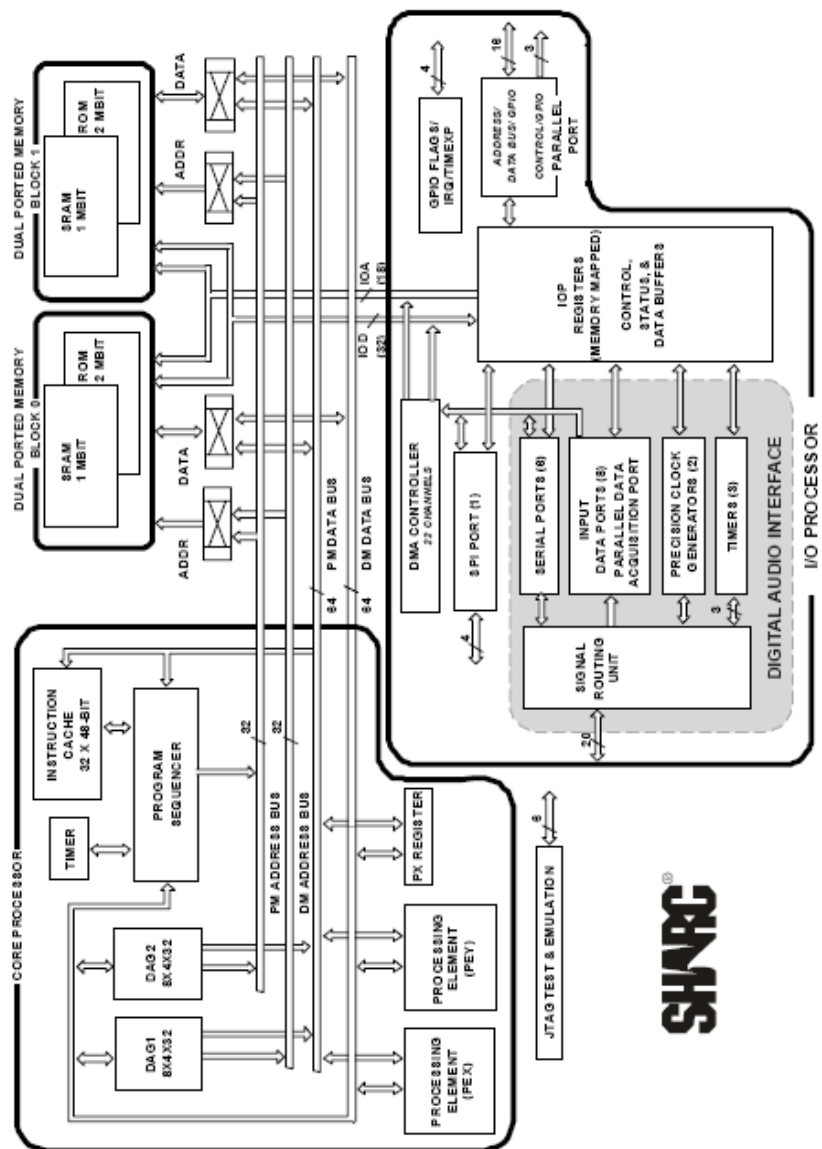


Figure 8 - Analog Devices SHARC 21261 internal architecture

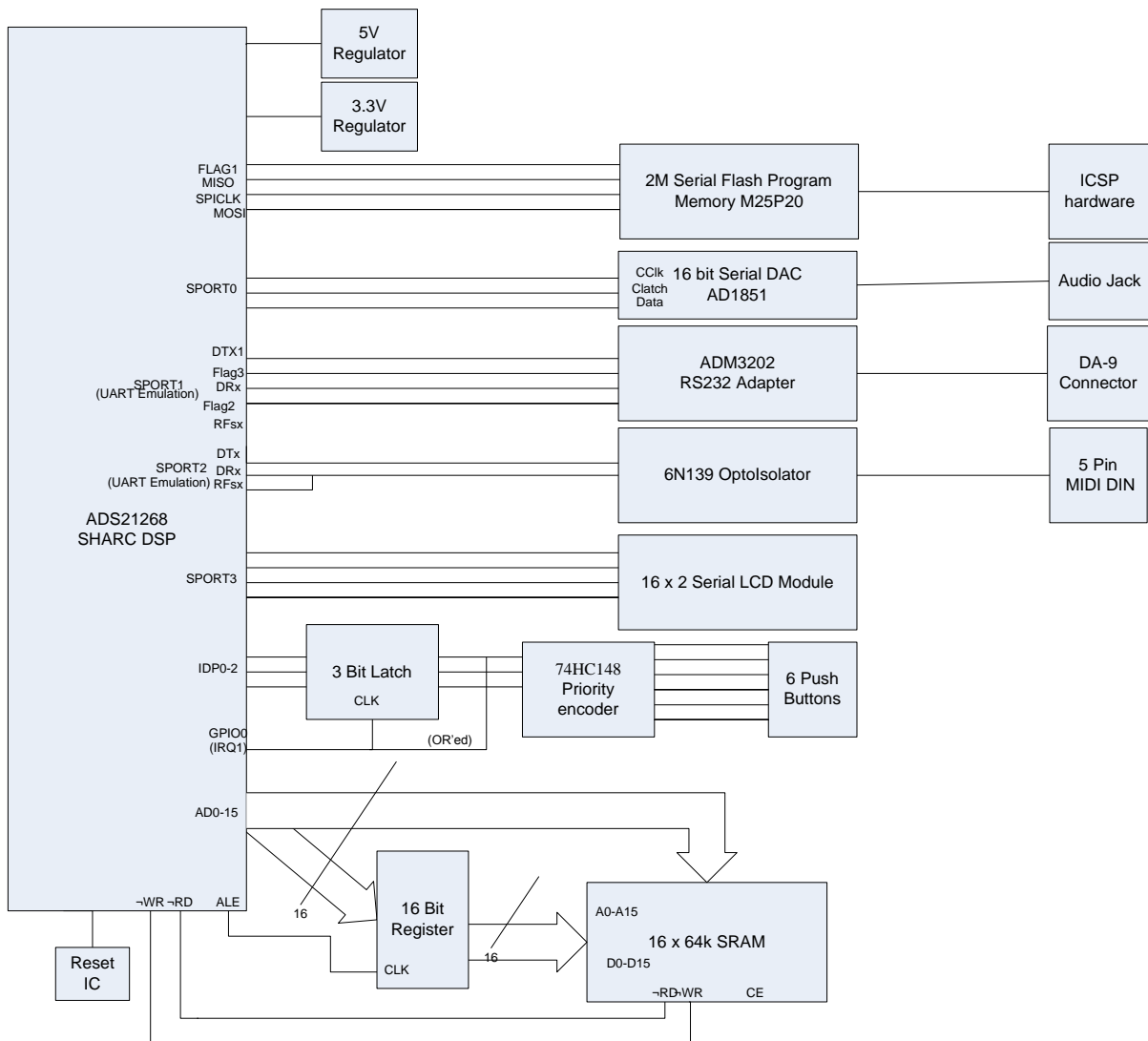


Figure 9 - System architecture

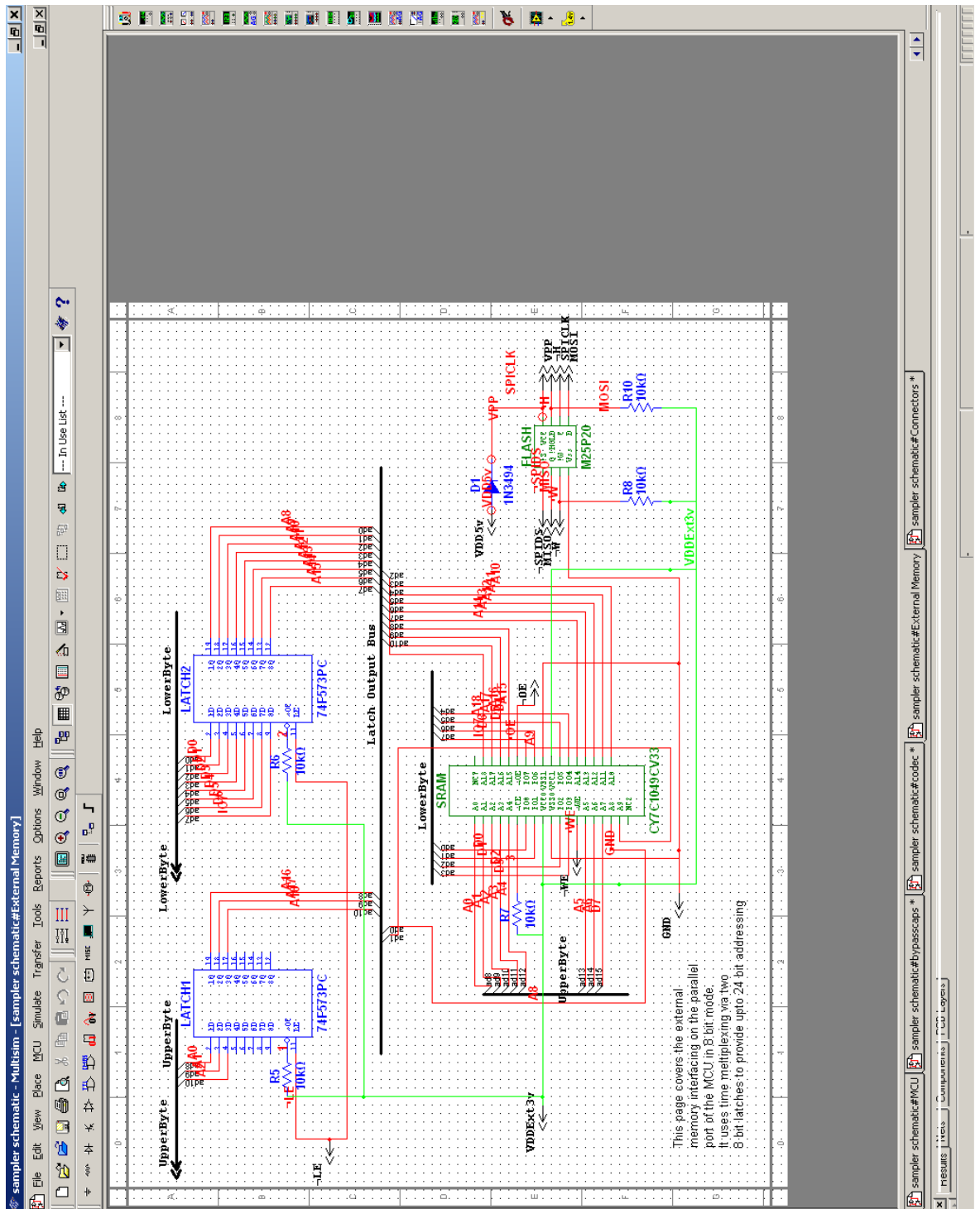


Figure 11 – Memory Interfacing Schematics

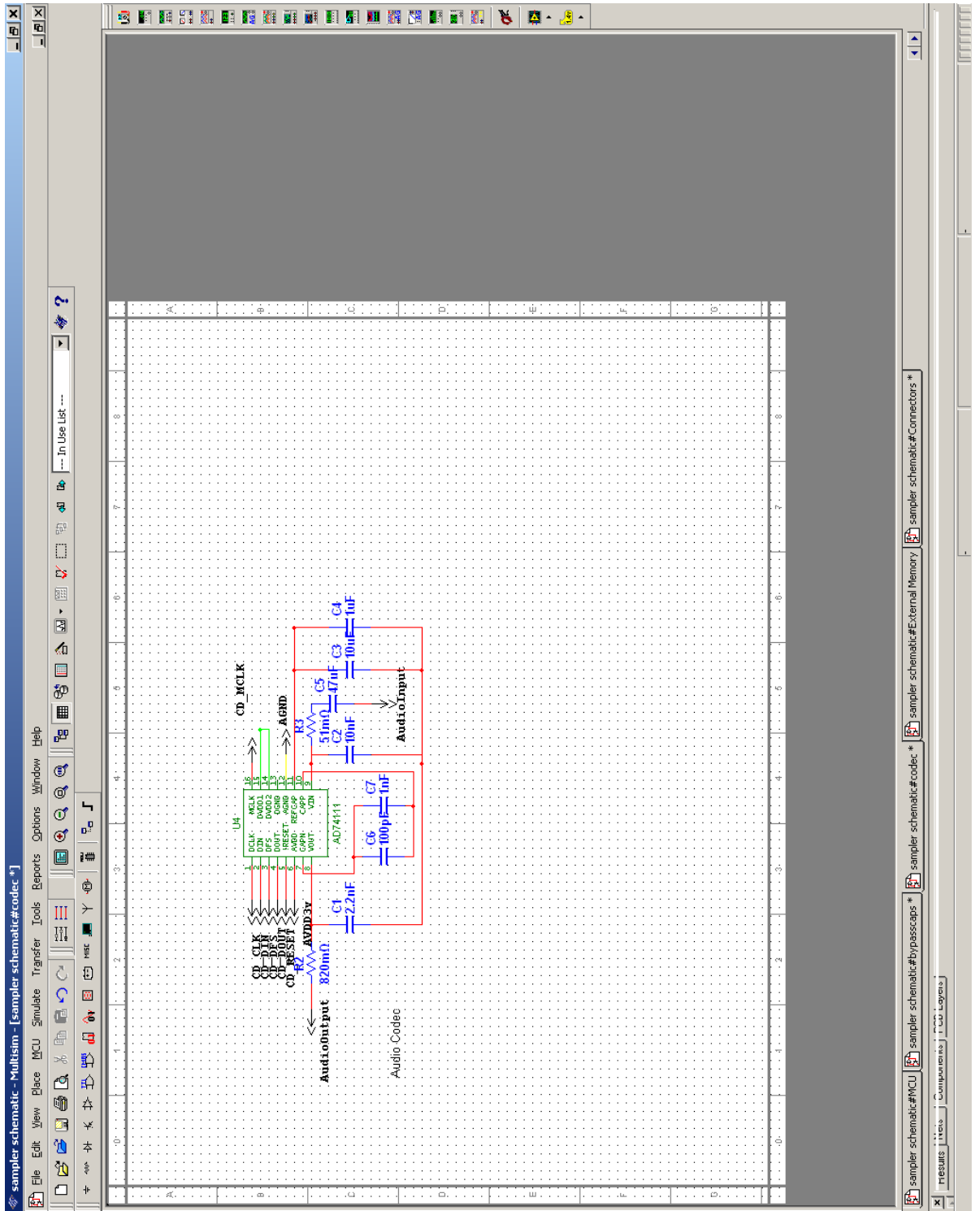


Figure 12 – Audio Codec Schematics

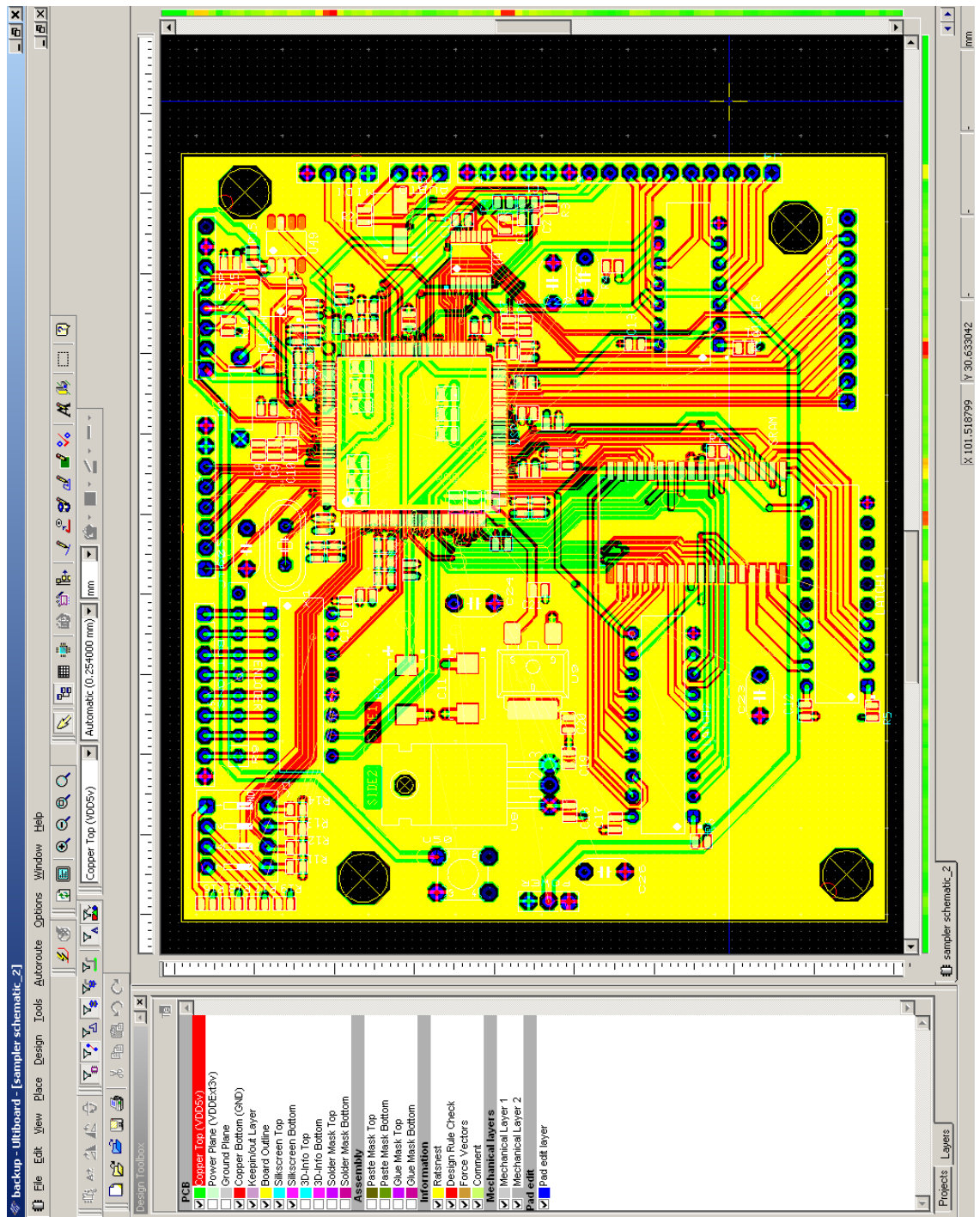


Figure 14 - PCB design

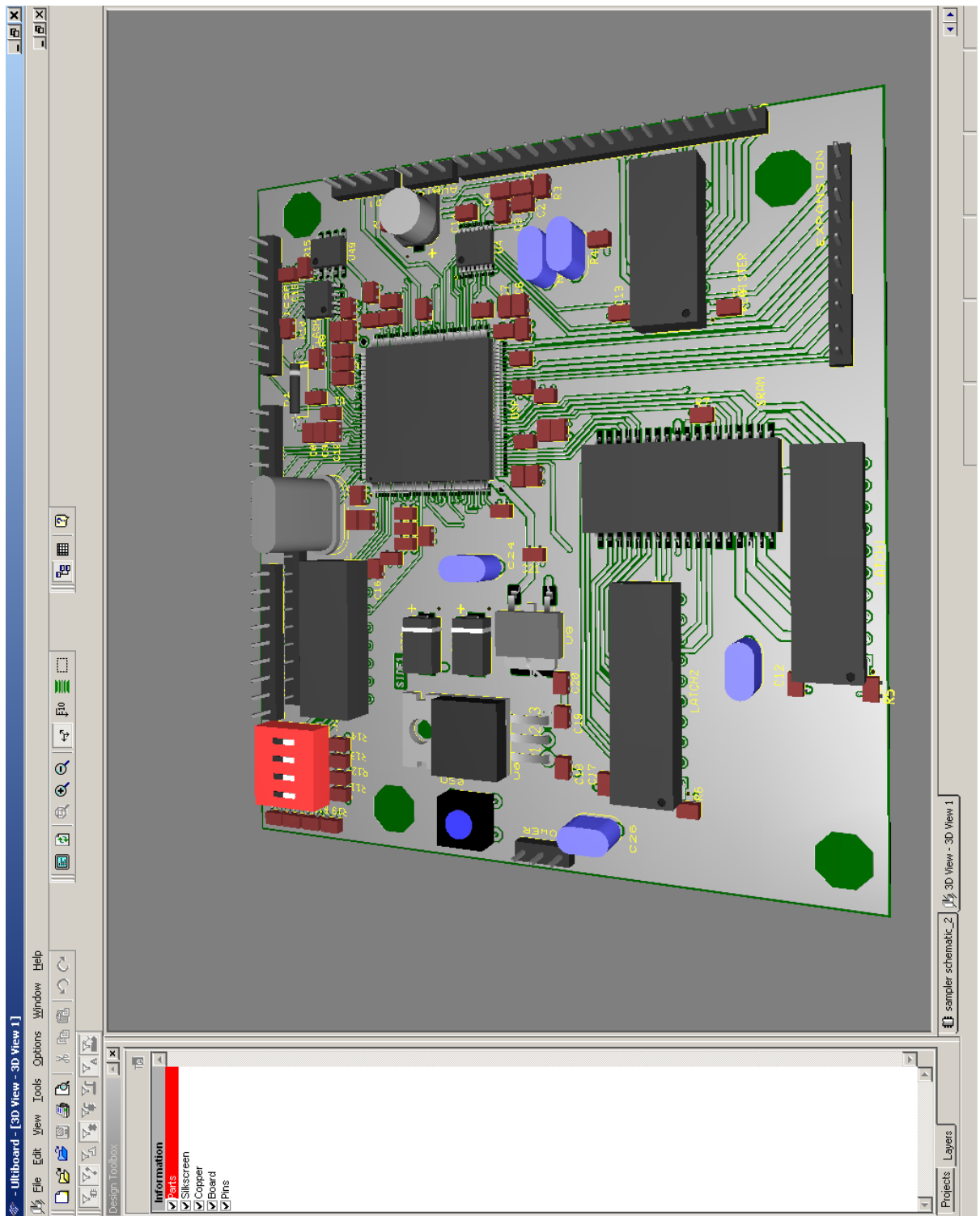


Figure 15 - 3D impression of assembled PCB



Figure 16 - Photo of assembled main PCB

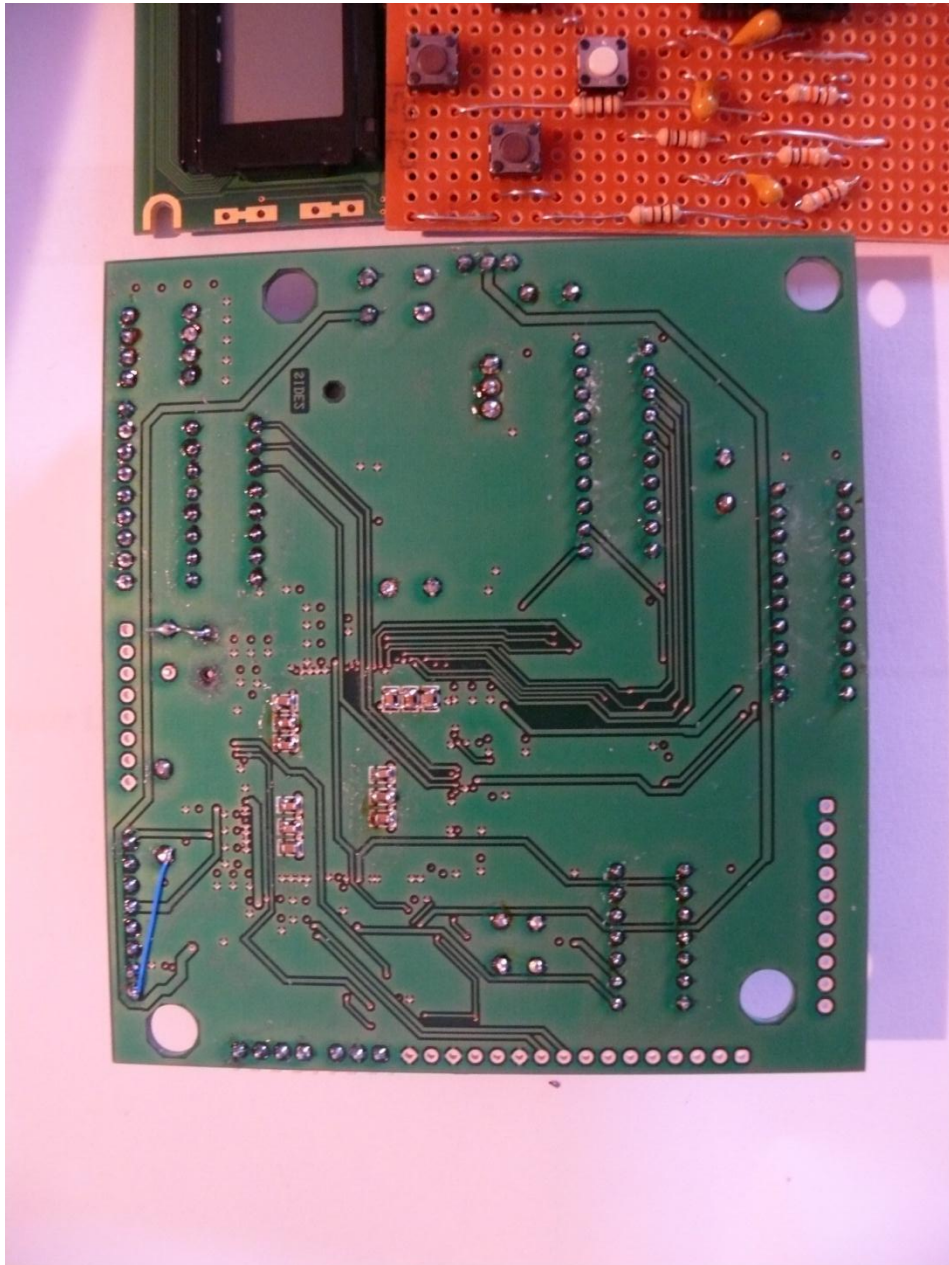


Figure 17 - Photo of interface board

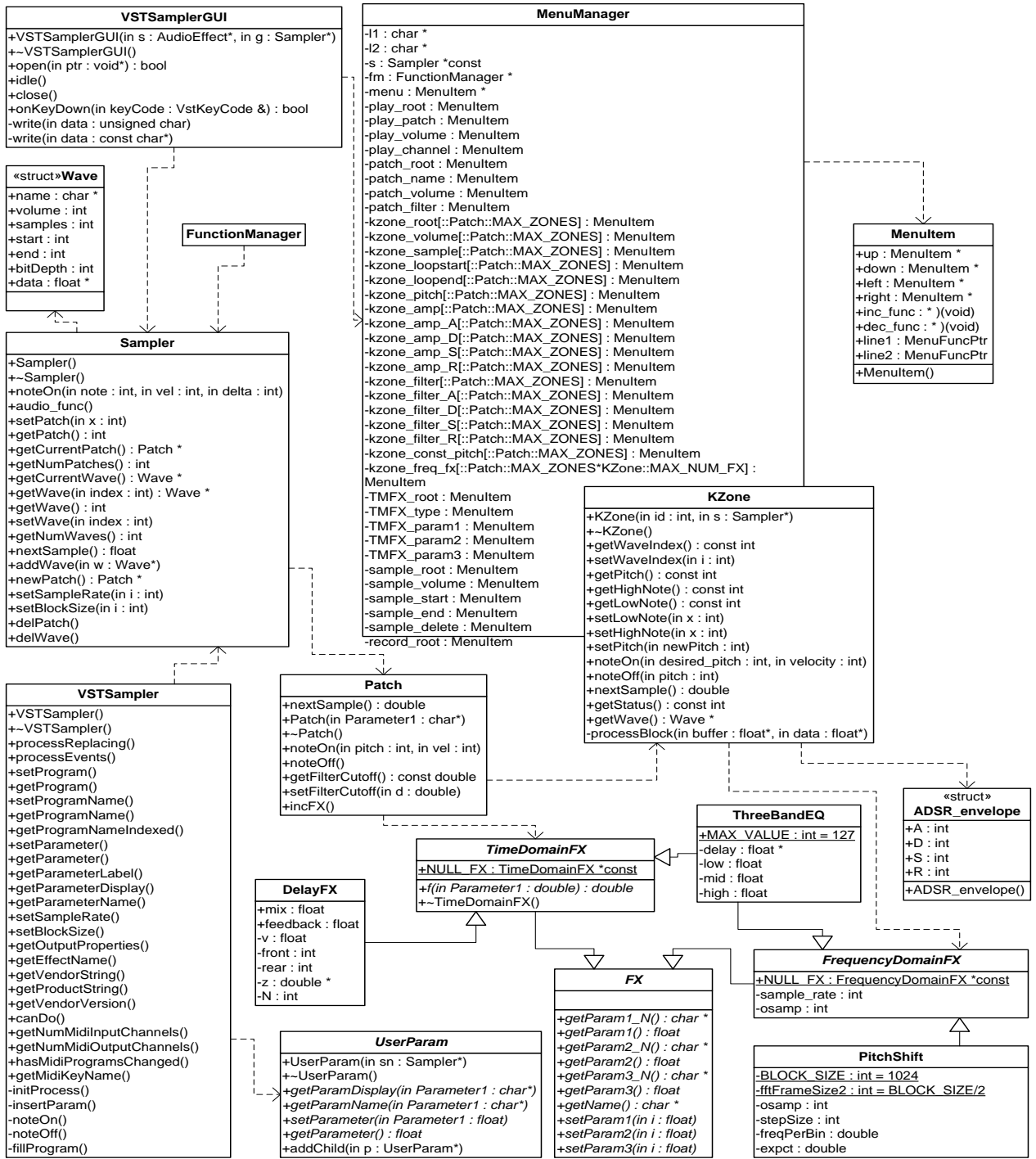


Figure 18 - Class Diagrams

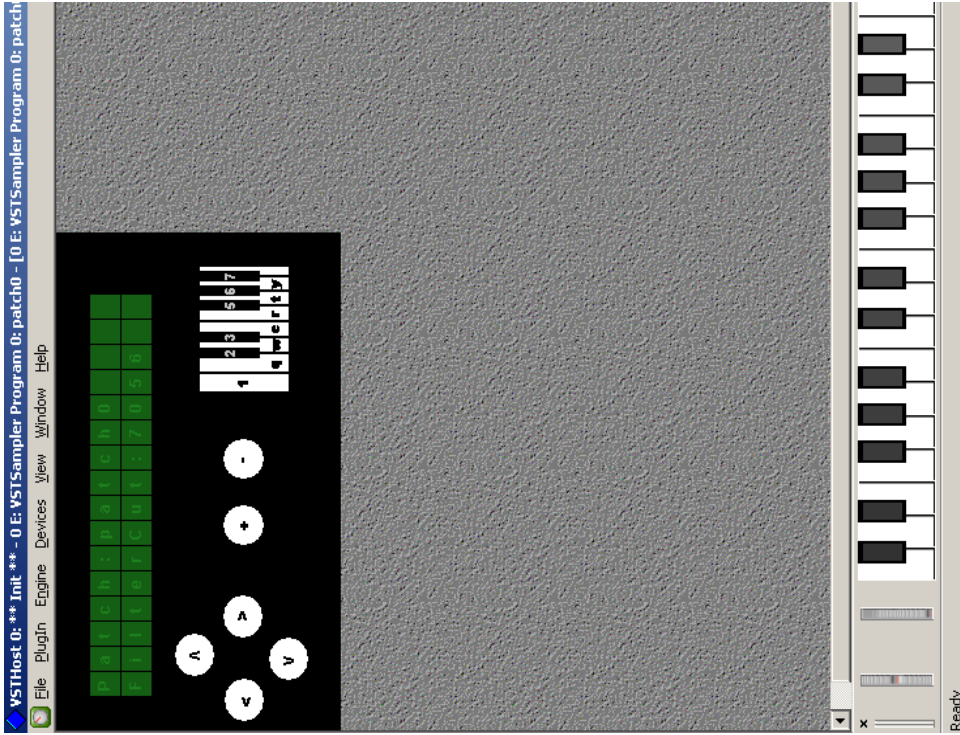


Figure 19 - Screenshot of VST GUI

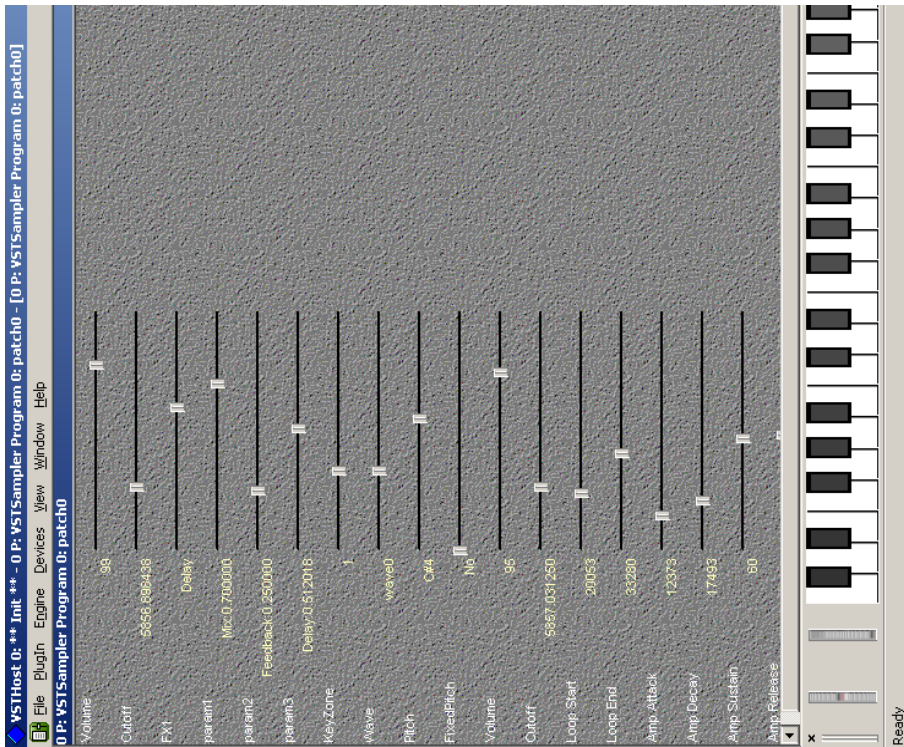


Figure 20 - Screenshot of VST Parameter Editor



Figure 21 - Screenshot from Waves API 2500 Compressor VST Emulation



Figure 22 - Photograph of API 2500 Hardware Compressor