

HELSINKI UNIVERSITY OF TECHNOLOGY

Department of Electrical and Communications Engineering
Laboratory of Acoustics and Audio Signal Processing

Jari Kleimola

Design and Implementation of a Software Sound Synthesizer

Master's Thesis

submitted in partial fulfillment of the requirements for the degree of
Master of Science in Technology.

Espoo, August 31st, 2005

Supervisor and Instructor: Professor Vesa Välimäki

Author:	Jari Kleimola		
Name of the Thesis:	Design and Implementation of a Software Sound Synthesizer		
Date:	August 31 st , 2005	Number of Pages:	96+4
Department:	Electrical and Communications Engineering		
Professorship:	S-89 Acoustics and Audio Signal Processing		
Supervisor:	Professor Vesa Välimäki		
Instructor:	Professor Vesa Välimäki		
<p>Increased processing power of personal computers has enabled their use as real-time virtual musical instruments. In this thesis, such a software sound synthesizer is designed and implemented, with the main objective being in the development of a composite synthesis architecture comprising several elementary synthesis techniques.</p> <p>First, a survey of sound synthesis, effects processing and modulation techniques was conducted, followed by an investigation to some existing implementations in hardware and software platforms. Next, a formal object-oriented design methodology was applied to capture the requirements of the implementation, and an architectural design phase was carried out to ensure that the requirements were fulfilled. Thereafter, the actual implementation work was divided between the reusable application framework library and the extended implementation packages. Finally, evaluation of the results was made in form of sound and source code analysis.</p> <p>As a conclusion, the composite synthesis architecture was found to be relatively intuitive and realizable. The generic object-oriented design methodology applied appeared to be well suited to the design of sound synthesis systems in general, but was considered to be too laborious to follow in every detail. The implementation work benefitted from the properly done design phase, however. The relative amount of man machine interface code compared to other subsystems was still surprisingly large. The timbral dimension of the realizable sound palette appeared to be quite wide, and the quality of the audio output was comparable, or even better than that of the existing implementations.</p>			
<p>Keywords: audio effects, musical acoustics, object-oriented design methods, software framework, sound synthesis</p>			

Tekijä:	Jari Kleimola		
Työn nimi:	Ohjelmistopohjaisen äänisyntetisaattorin suunnittelu ja toteutus		
Päivämäärä:	31. elokuuta 2005	Sivumäärä:	96+4
Osasto:	Sähkö- ja tietoliikennetekniikka		
Professuuri:	S-89 Akustiikka ja äänenkäsittelytekniikka		
Työn valvoja:	Professori Vesa Välimäki		
Työn ohjaaja:	Professori Vesa Välimäki		

Henkilökohtaisten tietokoneiden käyttö tosiaikaisina soitinsovelluksina on mahdollistunut lisääntyneen laskentakapasiteetin myötä. Tässä diplomityössä kuvataan em. kaltaisen ohjelmistopohjaisen äänisyntetisaattorin suunnittelu- ja toteutusprosessi, jonka päätaivoitteena oli useista eri perussynteesitekniikoista koostuvan synteesiarkkitehtuurin kehittäminen.

Työssä läpikäydään lukuisia äänisynteesi-, muokkaus- ja modulaatiotekniikoita, minkä jälkeen tarkastellaan joitakin jo olemassaolevia laitteisto- ja ohjelmistopohjaisia järjestelmiä. Toteutettavan sovelluksen vaatimusmäärittelyyn ja tätä seuraavaan järjestelmäarkkitehtuurin suunnitteluvaiheeseen sovellettiin yleiskäyttöistä oliopohjaista suunnittelumetodiikkaa. Varsinainen toteutusvaihe pilkottiin sovelluskehityksen ja sen varaan rakennetun laajennusosion kesken. Työn tulosten arviointiin käytetään ääni- ja lähdekoodianalyysia.

Työssä kehitetty kokoava synteesiarkkitehtuurirakenne osoittautui intuitiiviseksi ja toteutuskelpoiseksi ratkaisuksi. Käytetty suunnittelumetodiikka soveltui hyvin äänisynteesijärjestelmien suunnitteluun, mutta sitä pidettiin liian työläänä menetelmänä lähinnä ylläpitovaatimusten vuoksi. Toteutusvaiheessa hyvin tehdystä suunnittelusta oli luonnollisesti hyötyä, vaikkakin käyttöliittymäkoodin suhteellisen suuri määrä aiheutti analyysivaiheessa yllätyksen. Syntetisaattorin tuottaman äänimateriaalin monipuolisuus oli positiivinen havainto, minkä lisäksi äänen laatu osoittautui vertailukelpoiseksi, ellei jopa parempitasoiseksi kuin vastaavissa kaupallisissa sovelluksissa.

Avainsanat: musiikkiakustiikka, ohjelmistokehys, oliopohjainen suunnittelu, ääniefektit, äänisynteesi

Acknowledgements

I want to thank Professor Vesa Välimäki for supervising and instructing this thesis. Without his help this thesis would still be just a growing pile of papers and random notes, so thanks for helping me to see the big picture. His comments on both scientific and grammatical issues were also of great value.

I would also like to thank my employer Bombardier Transportation for lending me some extra time to finish this work.

Finally, I want to thank my parents Arja and Veikko, and my brother Jyrki for the support that I know I can count on. I want to express my deepest and warmest thank you to Tarja. I noticed, and I love you too.

Tapiola, August 31, 2005

Jari Kleimola

Table of Contents

1	INTRODUCTION	1
1.1	OBJECTIVES.....	1
1.2	SCOPE.....	2
1.3	STRUCTURE OF THE THESIS	2
2	SOUND SYNTHESIS CONCEPTS	4
2.1	SYNTHESIS TECHNIQUES	4
2.1.1	Additive Synthesis.....	4
2.1.2	Subtractive Synthesis	6
2.1.3	Modulation Techniques.....	7
2.1.4	Waveshaping	10
2.1.5	Karplus-Strong (Ks) Algorithm	13
2.1.6	Wavetable Synthesis.....	14
2.1.7	Synergy of Synthesis Methods	15
2.2	EFFECTS	16
2.2.1	Time-based Ambience Effects	17
2.2.2	Modulation Effects	17
2.2.3	Filters.....	18
2.2.4	Dynamics and Gain Control	18
2.2.5	Stereo and Panning.....	19
2.3	CONTROL SIGNALS.....	19
2.3.1	Internal Modulation.....	19
2.3.1.1	<i>Envelope Generators (EGs)</i>	19
2.3.1.2	<i>Low Frequency Oscillators (LFOs)</i>	20
2.3.1.3	<i>More Exotic Modulators</i>	20
2.3.2	Performance Control	20
2.3.3	Modulation Matrix	21
3	SOUND SYNTHESIS IN PRACTICE	23
3.1	MIXING AUDIO AND CONTROL SIGNALS	23
3.2	HARDWARE IMPLEMENTATIONS	24
3.3	SOFTWARE-BASED APPROACH	25
3.3.1	Synthesis Languages	25
3.3.2	Virtual Desktop Studio	26
3.3.3	Plugin Architectures	26
3.3.4	Internal Structure of Plugins.....	27
3.3.4.1	<i>Audio Processing</i>	28
3.3.4.2	<i>MIDI Event Processing</i>	29
3.3.4.3	<i>Parameter Handling</i>	30
3.3.5	Example Plugin Instrument	31
4	REQUIREMENTS SPECIFICATION	32
4.1	GENERAL DESCRIPTION.....	32
4.2	SYSTEM CONTEXT	33
4.3	USE CASES	34
4.4	CONCEPTUAL OBJECT MODEL.....	35
4.4.1	Conceptual Class Model.....	35

4.4.2	Dynamic Model.....	37
4.5	SYNTHESIS ARCHITECTURE AND PARAMETERS.....	37
4.5.1	Source Section.....	37
4.5.1.1	Particle.....	39
4.5.1.2	Modifier Block (MFX).....	42
4.5.2	Line Mixer Section.....	43
4.5.3	Master Mixer Section.....	44
4.5.4	Modulation Section.....	44
4.5.4.1	Sources.....	45
4.5.4.2	Destinations.....	45
4.5.4.3	Modulation Matrix.....	46
4.6	SPECIFIC REQUIREMENTS.....	46
4.6.1	Communication Interface Requirements.....	46
4.6.1.1	MIDI.....	47
4.6.1.2	ASIO.....	47
4.6.2	Software Interface Requirements.....	48
4.6.2.1	Vst2 Host.....	48
4.6.2.2	Files.....	48
4.6.2.3	Registry.....	51
4.6.3	Man Machine Interface (MMI).....	51
4.6.4	Performance Requirements.....	52
4.6.5	Hardware Interface Requirements.....	53
4.6.6	Testing Requirements.....	53
4.6.7	Portability Requirements.....	53
4.6.8	Safety Requirements.....	54
5	ARCHITECTURAL DESIGN	55
5.1	SYSTEM CONTEXT.....	55
5.2	SYSTEM DESIGN.....	56
5.2.1	Subsystem Decomposition.....	57
5.2.2	Subsystem Responsibilities and Interfaces.....	58
5.2.3	Interaction Diagrams.....	58
5.2.4	Subsystem Division, Referral and Concurrency.....	59
5.3	COMMON DESIGN ISSUES.....	60
5.3.1	Handling of Boundary Conditions.....	60
5.3.1.1	Initialization.....	60
5.3.1.2	Termination.....	60
5.3.1.3	Failure.....	60
5.3.1.4	Mode-specific Behaviour and Mode Changes.....	61
5.3.2	Implementation and Testing Environment.....	61
5.3.3	Extended Design.....	61
5.4	DETAILED SUBSYSTEM DESCRIPTIONS.....	62
5.4.1	Core.....	62
5.4.2	Event Handling (EH).....	62
5.4.3	DSP.....	63
5.4.4	Parameters and Patch.....	65
5.4.5	MMI.....	66
5.4.6	Files.....	67
5.4.7	Settings and Utilities.....	68
6	IMPLEMENTATION	69
6.1	CORE.....	69
6.2	EVENT HANDLING.....	69
6.2.1	MIDI Input Stream Processing.....	69
6.2.2	MIDI Processing at Audio Rate.....	71
6.2.3	Time Slicing Algorithm.....	72
6.3	DSP.....	73
6.3.1	Oscillators.....	73
6.3.2	Modifiers.....	76
6.3.3	Containers.....	78

6.3.4	Control Rate Modulation.....	78
6.3.5	Audio Rate Modulation.....	80
6.4	MMI.....	81
6.5	PARAMETERS, PATCHES AND FILES.....	82
6.6	SETTINGS AND UTILITIES.....	83
7	EVALUATION OF RESULTS	85
7.1	SOUND ANALYSIS.....	85
7.1.1	Sawtooth Waveforms	85
7.1.2	FM-style Electric Piano	87
7.2	PERFORMANCE MEASUREMENTS.....	89
7.3	SOURCE CODE ANALYSIS.....	90
8	CONCLUSIONS AND FURTHER WORK	91
8.1	CONCLUSIONS	91
8.2	FURTHER WORK.....	92
9	REFERENCES	93
10	APPENDIX A -- PARAMETERS	97

List of Abbreviations

AD/DA	Analog to Digital / Digital to Analog (converter)
ADSR	Attack / Decay / Sustain / Release
AM	Amplitude Modulation
API	Application Programming Interface
ARM	Audio Rate Modulation
ASIC	Application Specific Integrated Circuit
ASIO	Audio Stream Input Output
AU	Audio Unit
BPF	Band Pass Filter
BRF	Band Reject Filter
CPU	Central Processing Unit
CRM	Control Rate Modulation
DAW	Digital Audio Workstation
DCA	Digitally Controlled Amplifier
DCF	Digitally Controlled Filter
DCO	Digitally Controlled Oscillator
DCW	Digitally Controlled Waveshaper
DLL	Dynamic Link Library
DSF	Discrete Summation Formulae
DSP	Digital Signal Processing / Processor
DXi	DirectX instrument
EG	Envelope Generator
EQ	Equalizer
EXT	External audio
FIR	Finite Impulse Response
FM	Frequency Modulation
FPU	Floating Point Unit
HPF	High Pass Filter
IDFT	Inverse Discrete Fourier Transformation
IIR	Infinite Impulse Response
IO	Input/Output
LAN	Local Area Network
LFO	Low Frequency Oscillator
LPF	Low Pass Filter
MIDI	Musical Instrument Digital Interface
MME	MultiMedia Extensions
MMI	Man Machine Interface
MS	Main - Side stereo signal format
OS	Operating System
OSC	Oscillator
PC	Personal Computer
PCM	Pulse Coded Modulation
PD	Phase Distortion
RM	Ring Modulation

RVG	Random Value Generator
SDK	Software Development Kit
SNR	Signal to Noise Ratio
STL	Standard Template Library
SSE	Streaming Single-instruction-multiple-data Extensions
UC	Use Case
UML	Unified Modeling Language
VCA	Voltage Controlled Amplifier
VCF	Voltage Controlled Filter
VST	Virtual Studio Technology
XOR	Exclusive or

List of Symbols

A_c, A_m	amplitude of carrier and modulator
A_{max}	maximum amplitude
$A_g(n)$	global amplitude
$A_k(n)$	amplitude of oscillator k
d_N	polynomial coefficient
f_0, f_c, f_m	frequency (in hertz) of fundamental, carrier and modulator
f_s	sampling frequency
$F_k(n)$	frequency of oscillator k
$F(g)$	waveshaper
$g(n)$	source signal
I	modulation index
$J_k(I)$	Bessel function of the first kind
k	partial number, polynomial order
M	number of oscillators
n	sample number
N	highest harmonic, wavetable size
\mathbf{p}	point (x, y)
r	spectral asymmetry factor
s	stretch factor
$s(n)$	output signal
SI	sampling increment
$T_k(x)$	Chebyshev polynomial of the first kind
$x(n)$	input signal
$y(n)$	output signal
y_{max}	maximum y coordinate
α	distortion index
λ	wavelength
ω, ω_c	frequency (in radians per second), carrier frequency
$\theta_k, \theta_k(n)$	phase of oscillator k
θ_c, θ_m	carrier phase, modulator phase

Chapter

1 Introduction

1.1 Objectives

I was introduced to the synthetic sound at a time when the first affordable digital synthesizers started to make their debut at local music stores. Ever since I have been active both as a performer and as a sound designer, and managed to tweak parameters on dozens of different models and brands of synthesizers, both analog and digital alike. I have often wondered what would my personal dream machine be like, i.e. what set of parameters would it have, how would it be programmed and played, and most importantly, what would it sound like (or preferably — unlike). This thesis tries to find answers to these questions.

Having also investigated a wealth of different synthesis techniques, I have noticed that each technique has its strong and weak points in terms of sounds that it can produce. Would it then be possible to integrate those seemingly unrelated methods of sound generation together, under single composite conceptual model, and strengthen the weak parts of one technique with another more suitable method ?

Modular synthesizer environments allow already that flexibility, but their user interfaces are often so versatile, that it is easy to lose the timbres into the forest. A semi-modular synthesis architecture proposed in this thesis provides a considerable amount of freedom when patching the instrument, but has still enough rigidity so that more organized interaction mechanisms can be utilized. A novelty concept also discussed later is the hierarchical patch structure, where composite patch is constructed from a collection of smaller subpatch components. This facilitates sound design at different abstraction levels, and as any individual parameter can be overridden at higher levels, a basic sound component library can be reused at will. Furthermore, this basic library could be even built from the pre-programmed synthesizer patch banks of earlier years, thus recycling (albeit in approximate form) those proven sounds in a new context.

Programming (or patching) a synthesizer is only one side of the story, however. An equally important part is the responsiveness of the instrument in real-time performance situations. The large amount of synthesis parameters, and the potentially minor audible effect of a single parameter tweak do cause a controlling problem. A partial solution to

this is proposed in form of a macro facility, i.e. the grouping of several synthesis parameters under a single controller. Such a macro could for example be bound to a single slider to control an abstract ‘brightness’ attribute of the composite sound.

In summary, the problem space is quite large, and does therefore require a formal design methodology to be followed in order to be manageable. A widely used object-oriented requirement analysis and architectural design methodology is utilized, and its suitability to the particular project shall be discussed. Observations on the implementation phase shall also be made. In particular, the amount of synthesis related code vs. supporting (trivial) coding shall be compared, and the framework expandability, code reusability and complexity shall be discussed.

1.2 Scope

This thesis shall take a practical approach to sound synthesis. Theoretical background is presented, but mathematical treatment is kept in minimum and left to numerous reference sources readily available. Also, the amount of program code is so large that it would not make sense to include every bit of it in this thesis, not even inside the appendices. Some of the key algorithms are described with pseudo code snippets, though.

Synthesizer implementation realized in this work should be regarded as a working prototype of a final product, and any functional difference (usually a lack of feature) between the two is stated in relevant context. The prototype shall provide enough features so that the objectives described above are reached, and so that quantitative and qualitative results can be evaluated against realizability of the final product.

A prior knowledge of hardware or software synthesizers, effect devices nor musical sound production environments is not required of the reader. It is assumed though, that he has some background on digital signal processing and some familiarity on object-oriented design and programming. Design diagrams are presented using the UML notation, and pseudo code snippets are given in a language similar to C++.

1.3 Structure of the Thesis

Chapter 2 gives the theoretical treatment of sound synthesis concepts. It starts by first describing some of the techniques that are available for the creation of audio signals, and continues by examining various means of processing them. Discussion of control and modulation signals follows thereafter.

Chapter 3 moves from concepts to practice, by first investigating the problems encountered when audio and control signals are combined into a working synthesizer unit. Section 3.2 discusses briefly existing hardware based solutions and compares the architectural constructions of three flagship workstations from Roland, Yamaha and

Korg. Section 3.3 describes software based approaches with a more detailed view of VST plugins

Chapters 4 and 5 carry out the formal design phases of the project. Requirements are captured in chapter 4, starting from the user's view of the system, and reflecting that directly to the software requirements of the work. Sections 4.2 through 4.4 give a detailed model description of a general software synthesizer, which can be applied to any synthesizer architecture. The synthesis architecture of PHUZOR, which is the synthesizer implemented in this work, is given in section 4.5 as an extension of the generic model. The MMI subsystem is defined in section 4.6 among other specific requirements.

The architectural design phase is described in chapter 5 so that all requirements given in chapter 4 are implemented. It takes a topdown approach by decomposing the synthesizer into packages, subsystems, classes, attributes, methods and their interfaces. The purpose of design phases is to describe what needs to be done, and leave algorithmic issues to the actual implementation phase. This is selectively discussed in chapter 6.

Chapter 7 evaluates the results of the work, and a conclusion is finally drawn in chapter 8.

Chapter

2 Sound Synthesis Concepts

The signal routing mechanism of a typical synthesizer categorizes processed signals into three functional groups. Audio signals are born inside the oscillating components, are further processed by effect units, and are eventually sent out of the box to make audible sound. Modulation signals shape these audio signals during their journey out by continuously changing sound's loudness, pitch and timbral properties. The third group consists of control signals that are results of performer's gestures. They are used to generate gates, triggers and other expression events that are used in a manner similar to that of the modulation group.

In fact, all these signals are basically alike, as they can be simply treated as functions of time. Their data rates are different however, because audio signals reach frequencies that go up to 20 kHz and beyond, and according to the sampling theorem, must be handled in the digital domain at least twice that frequency. Modulation and control signals do not need to be updated at such a high rate, and control rates below 100 Hz are not uncommon.

Discussion of audio rate signals is presented in sections 2.1 and 2.2, followed by control rate signals in section 2.3.

2.1 Synthesis Techniques

Because of time and space limitations, a special consideration is given to the techniques that are utilized in this thesis. More thorough discussion can be found in [1] - [4].

2.1.1 Additive Synthesis

In additive synthesis elementary waveforms are fused together into a more complex composite waveform. Component waveforms are usually simple (traditionally sinusoidal, in order to have precise control over (in)harmonic content of the resultant spectrum), although waves of any complexity can be used. Figure 2.1 in the following page shows a simple additive instrument with M sinusoidal oscillators, each having inputs for frequency, amplitude and phase. Oscillator outputs are summed together and scaled so that the maximum amplitude of the composite sound is equal to A_{\max} .

If oscillator inputs F_k , A_k and θ_k remain constant over time, a waveform with a static spectrum is produced. This method can be used in digital oscillators when generating classic analog style waveforms, like square or sawtooth waves, as it offers a way to define precise harmonic structure and guaranteed bandlimited spectrum. For example, a sawtooth wave can be approximated by summing all the harmonics having relative amplitude coefficients $A_k = 1/k$ and a square wave by summing only odd harmonics with amplitudes $1/k$.

A dynamic spectrum can be obtained when time variant coefficients are introduced, and according to the Fourier theorem, any periodic signal can be constructed from infinite sum of sinusoidal components having specific frequency, amplitude and phase. In approximate form this can be written as [3]

$$s(n) = A_g(n) \sum_{k=1}^M A_k(n) \sin \left[(k\omega n + 2\pi F_k(n)) + \theta_k(n) \right] \quad (2.1)$$

where $s(n)$ is the output signal, n is a sample number, ω is the radian fundamental frequency of the note, k is the partial number, M is the number of oscillators, $A_k(n)$ is the amplitude, $F_k(n)$ the frequency and $\theta_k(n)$ the phase track of k^{th} partial, and $A_g(n)$ is the overall amplitude.

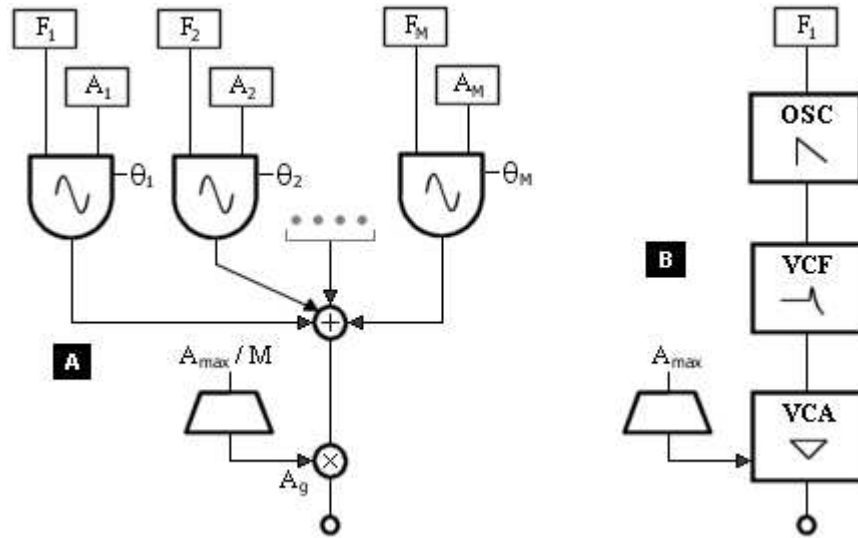


Figure 2.1 a) Additive synthesis instrument with global amplitude scaling. b) Subtractive synthesis instrument.

Amplitude, frequency and phase tracking does not have to occur at audio rate, so modulation signals can be approximated by linear or exponential curves of control rate. However, when large number of partials are to be controlled, a lot of control data is still needed. A variant method called group additive synthesis reduces the amount of control parameters by mixing partials that share a similar temporal loudness and pitch evolution into a single wavetable [5]. These wavetables are then used instead of simple sinusoidal oscillators in order to produce more efficient way to create a composite waveform, but that is achieved at the cost of generality. Another approach is the wavestacking

technique utilized in the commercial sampler units, where oscillator source waveforms are actually complex multicycle snapshots of acoustic instrument timbres [1].

2.1.2 Subtractive Synthesis

This technique was used in early analog synthesizers, and is therefore sometimes misleadingly termed as analog synthesis. Subtractive synthesis starts with spectrally rich source material, such as noise, sawtooth, or pulse waveforms. Desired output signal is then formed by removing excess spectral content through filtering, and possibly by emphasizing certain formant frequencies. The amplifier component at the end of the signal chain allows global amplitude scaling (see Figure 2.1). In a sense, the subtractive process can be seen as a rudimentary model of an acoustic instrument, where the source oscillator acts as the excitation component, and the filter as a resonator.

However, key strengths of subtractive technique does not lie in providing convincing emulation of acoustic instrument sounds, due to over-simplified model of excitation and resonator. Rather, it has sculpted sounds that are nowadays considered as sounds having character in their own right, and has become a source of emulation itself. It is also a relatively intuitive model of synthesis, as OSC-VCF-VCA architecture is commonplace even in digital synthesizers utilizing different synthesis methods.

Oscillator (OSC)

Raw source material is produced by one or more oscillators. Analog circuitry is not able to conveniently produce any waveform imaginable, so the basic waveset of subtractive oscillators include sources for sine, triangle, square, pulse (with adjustable duty width), sawtooth and noise waveshapes. Bandlimited forms of these waves can be generated by additive method described above, for other solutions see [6].

Noise waves are aperiodic, and as such do not have a distinct pitch. They can be parametrized by the amount of energy present in higher frequencies, and are associated with a color analogously with visible spectrum of light. White noise has equal power density throughout the entire frequency range, and other shades of noise can be generated by running it through a low pass filter. In digital domain, noise can be produced using one of the pseudo random number generator algorithms.

If two oscillators are at disposal, a slight detuning can be used to add animation to the resultant sound. Ring modulation and audio rate frequency modulation between oscillators are also commonly used methods for producing a wider source material palette. Another characteristic subtractive sound can be made with hard sync, where slave oscillator's cycle is reset whenever master oscillator has completed the cycle of its own [7]. The resultant sound is spectrally rich, and thus subjective to aliasing in digital domain, alas.

Filter (VCF)

The sound produced by a subtractive system is characterized mostly by the type and quality of filtering components in use. Many a synthesizer has become a legend or total failure due to the filtering implementation, the famous example of the first kind being the Moogs with their imperfect but warm sounding ladder filter network [8]. In general, the subtractive technique utilizes filters operating in four basic modes, which can be categorized by their frequency response properties. A low pass filter (LPF) is the most commonly used mode designed to pass frequencies below a specific cutoff frequency, and to attenuate those above it. Other common types are high pass (HP), band pass (BP) and band reject (BR) modes.

The cutoff or center frequency is adjustable, and a narrow band of frequencies around that point can be boosted by an additional resonance parameter. The amount of attenuation in stop band is dependent on filter design and the amount of poles in its transfer function, and is specified in dB / octave units (a 6 dB drop halves the amplitude, and one octave doubles the frequency). 6, 12 and 24 dB per octave slopes are most common, and correspond to 1, 2 or 4 pole designs, respectively. In frequency response graphs, a steeper curve means more attenuation.

Filters in different modes can be connected together to form a composite filter bank. For example BP and BR modes can be emulated by arranging LP and HP filters serially or parallel to each other. Another configuration is to use multiple BP and BR filters together to realize formant filters, which are basically filters with multiple resonance peaks throughout the passband.

Amplifier (VCA)

In practical implementations, the signal coming out of the filter circuitry must be brought up in level so that the output signal shall be strong enough to drive the line level inputs. This is done in an amplifier, and although not part of the actual synthesis process, it somewhat affects the color of produced signal because of non-linearities. It usually consists of a knob for setting the overall gain level, and one input and one output port. Most importantly, it has also a control input for temporal amplitude level changes, which is the point where amplitude envelope generator's ADSR signal shall be patched.

2.1.3 Modulation Techniques

Nonlinear synthesis techniques produce sound by distorting simple source waveform into sonically more complex structure, using only a handful of synthesis parameters along the process. They can be computed quite efficiently, and have been utilized in various commercial synthesizer products ever since the beginning of 1980's. Modulation synthesis is one of such techniques, in which some parameter of a carrier oscillator is continuously modified with a modulating oscillator. Having their roots in radio transmission techniques and on the other hand in musician's articulation techniques to add minor nuances into sound of his instrument (by introducing small amounts of vibrato or tremolo), they have been in existence for quite some time. However, their use

in producing complex audio waveforms is a relatively new discovery, and their power becomes apparent when both carrier and the modulator signal frequencies are brought into the audible frequency range.

In the following discussion, sinusoidal waves are used for both carrier and modulator. In practice, any signal can be used, but as even simple systems with just two sinusoidal oscillators are capable in producing very complex timbres, it is often not necessary to do so. A sine carrier signal is given as [9]

$$s(n) = A(n) \sin(n\omega_c + n\theta_c) = A(n) \sin(n2\pi f_c + n\theta_c) \quad (2.2)$$

where n is a sample number, $A(n)$ is the carrier amplitude, ω_c the carrier angular frequency, f_c is the carrier frequency in Hz, and θ_c the carrier phase offset. Possible modulation destinations in this equation are A , f_c and θ_c , which lead to amplitude, frequency and phase modulation, respectively. Latter two are collectively called angular modulation techniques.

$$s(n) = A(n) \sin(n2\pi f_c + n\theta_c)$$

AM ———┐
 FM ———┤
 PM ———┘

Amplitude modulation

In amplitude modulation, the amplitude of the carrier signal follows modulator's output signal, and can be implemented in the digital domain by multiplying outputs of both oscillators together. In cases where the modulator is unipolar, we speak of amplitude modulation (AM), and when bipolar, we speak of ring modulation (RM) [1]. In the sinusoidal AM, the carrier frequency is surrounded by two sidebands, lower at $f_c - f_m$ and higher at $f_c + f_m$ (for complex waves, each partial spreads around fundamental frequency). Amplitudes of both sidebands are equal, and proportional to the amplitude of the modulator. In RM, the resultant spectrum does not include carrier's fundamental frequency spectrum component, as its energy is transformed into the sidebands. In both AM and RM, the sidebands are bound to be in inharmonic relation to the carrier and modulator frequencies, and particularly RM has a characteristic metallic sound.

Basic logical functions (OR, XOR, AND) between source signals can be used instead of multiplication, but as the transformation operation is quite radical, the resulting signal aliases and overflows easily. Best results are achieved when source signals are closely related at amplitude and frequency levels, but any source waveform can be used as long as its amplitude level is kept small (this is acceptable as the resulting signal covers full amplitude range independently of source signal scaling).

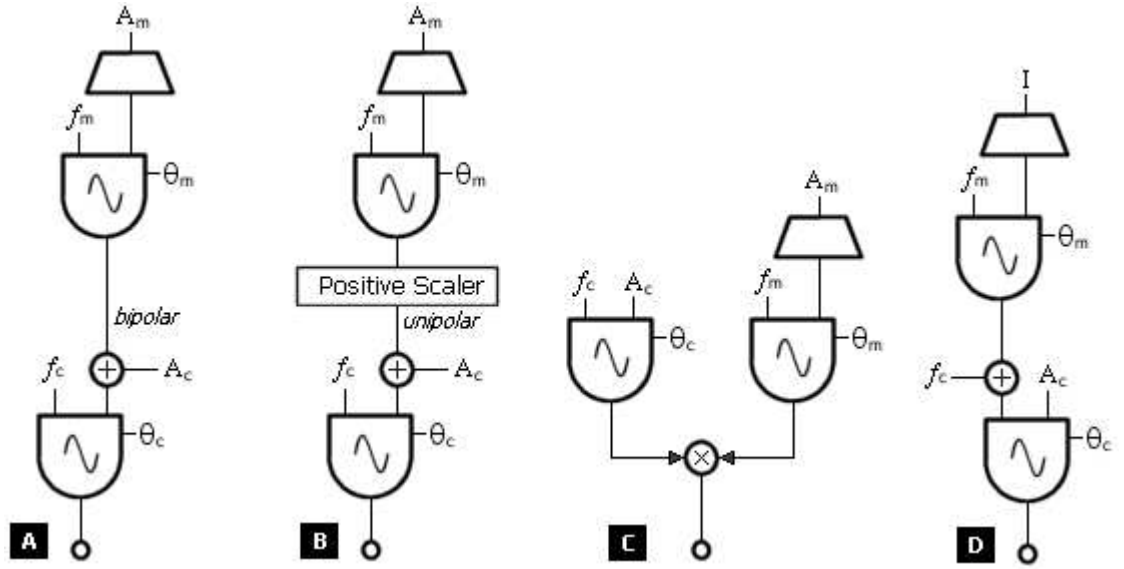


Figure 2.2 a) Ring Modulation instrument b) Amplitude Modulation instrument. c) Shows equivalent implementation of A. d) Frequency Modulation instrument.

Frequency and Phase modulation (FM and PM)

Although Chowning's classic paper [10] and Yamaha's marketing jargon both speak of FM, they actually are doing phase modulation. The difference between these methods is that in PM the phase of the carrier varies directly with the modulating signal, while in FM the phase is varied with the *integral* of the modulating signal [9]. In this discussion I shall adapt to the common nomenclature, and use the term FM to cover both angular modulation methods, and restrict the study to phase modulation.

Figure 2.2d shows a simple FM stack instrument. For simple FM with one sinusoidal modulator and one carrier, we can write equation 2.2 as

$$s(n) = A(n) \sin \left[2\pi f_c n + I \sin(2\pi f_m n) \right] \quad (2.3)$$

where I is the modulation index defining the bandwidth of the resulting signal. Increasing I spreads carrier energy to the sidebands that are located symmetrically around the carrier at $(f_c \pm f_m)$, $(f_c \pm 2f_m)$ and so on, with relative amplitude levels determined by Bessel functions of the first kind. This can be seen from equation 2.4, which is a generalized Fourier series form of equation 2.3 :

$$s(n) = A(n) \sum_{k=-\infty}^{\infty} J_k(I) \sin(2\pi f_c n + k2\pi f_m n) \quad (2.4)$$

where k is the order of Bessel function and a number of the sideband. Negative sidebands are reflected at 0 Hz with phase inversion, and summed to the corresponding sideband at positive frequency. If f_c / f_m can be presented as a ratio of two integers, a harmonic spectrum is produced, but if this is not the case, an inharmonic spectrum is resulted as the reflected components fall between frequencies $f_c \pm k f_m$.

Palamin et al. suggest an additional parameter r to equation 2.4, allowing control over spectral asymmetry around carrier frequency [11] :

$$s(n) = A(n) \sum_{k=-\infty}^{\infty} r^{|k|} J_k(I) \sin(2\pi f_c n + k2\pi f_m n) \quad (2.5)$$

When $r > 1$ amplitudes of sidebands higher than f_c are increased, and if $r < 1$ the lower end gets the boost, so r can be used as a filter type of effect. For synthesis equation (referring to Figure 2.2d), this results the term I to be multiplied by $(r + 1/r) / 2$, and the term A_c to be multiplied by $\exp(I (r - 1/r) \cos(2\pi f_m n) / 2)$. A dynamic spectrum can be achieved when I and/or r are modulated in the time domain.

There exists also other variations of simple FM scheme, namely double FM [12] (where the carrier of equation 2.3 is replaced with another modulator), the use of multiple carriers and modulators [1], using complex waves as modulators [13] and feedback FM [1] (where the output of oscillator is fed back to the system's frequency modulation input).

2.1.4 Waveshaping

Waveshaping is a general nonlinear synthesis technique. The classic articles describing it are [14] - [16], and some well-written tutorials can be found in [1], [2] and [17]. In waveshaping, instantaneous amplitude of a source signal $g(n)$ is distorted with a transfer function F to produce output $y(n)$, which can be written as

$$y(n) = F(g(n)) \quad (2.6)$$

If the graph of F is a straight line without any discontinuities, there is no distortion, and the outputted signal replicates the input (possibly with amplitude scaling, in respect to different slopes of F). However, if F is a nonlinear function, a change in input's amplitude causes a change in the output waveform's shape, denoting a spectral mutation. The choice of F is naturally most crucial parameter of the technique, in practical implementations constrained at $F(0) = 0$, and scaled to produce values in the same range as y and g . Any discontinuities or sharp edges produce unlimited amount of partials, which manifest themselves as aliasing. Bandlimiting is also difficult to achieve with complex input signals, so practical implementations often use a sinusoidal function as $g(n)$. Figure 2.3 shows sample outputs when $\sin(n)$ is shaped with various transfer functions.

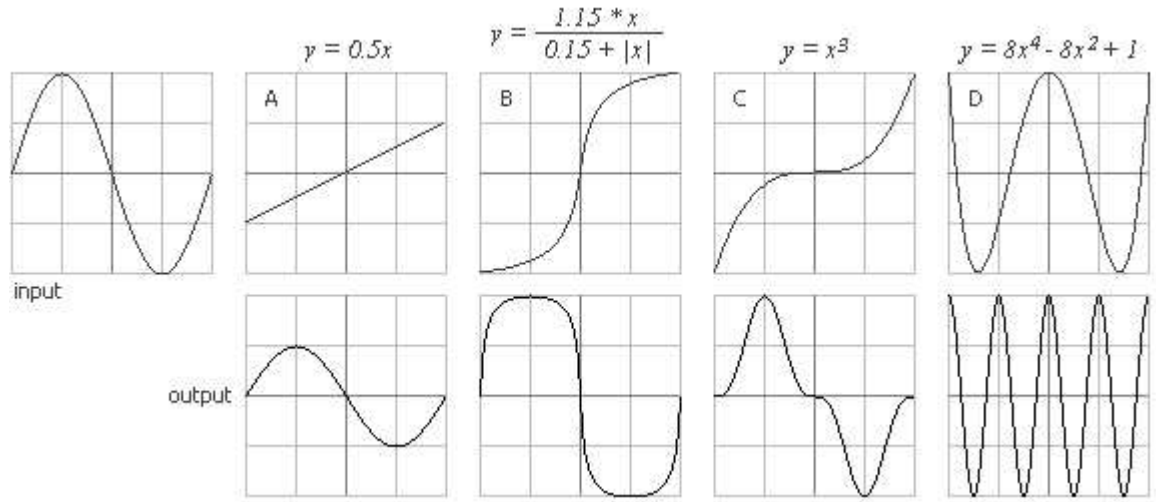


Figure 2.3 Wave shaping applied to a sinusoidal input signal. A) Attenuation. B) Compression with soft clipping. C) Odd transfer function produces only odd partials. D) Chebyshev polynomial T_4 .

Using polynomials as a transfer function

Polynomials have many useful properties when used as the transfer function. First, they provide a good approximation of any smooth curve, so a wide range of transfer functions can be represented in polynomial form. Second, the degree of the polynomial determines the highest harmonic that is produced when cosine source is fed through the waveshaper, thus bandlimiting the resultant output spectrum at will. Third, relative weights of partials are determined by polynomial's coefficients, which makes it possible to have precise control over the produced spectrum, and also dynamic control of all harmonics using a single parameter. This distortion index (α) is analogous to the modulation index used in FM synthesis. Introducing α into equation 2.6 and replacing $g(n)$ with $\cos(n)$ we can write [2, pp. 132-135]

$$y(n) = F(\alpha \cos(n)) = d_0 + d_1 \alpha x + d_2 \alpha^2 x^2 + \dots + d_N \alpha^N x^N \quad (2.7)$$

where d_i are the polynomial coefficients, N is the highest harmonic produced, and x is a cosine wave with unity amplitude. If coefficients are given, the amplitude of each harmonic can be calculated using Pascal's triangle with binomial coefficients and weighted by distortion index α , raised to the appropriate power.

Chebyshev polynomials of the first kind possess a property that makes them a worthy solution when synthesizing a static harmonic spectrum. If a cosine wave of unity amplitude and a frequency of f_0 is passed through transfer function consisting of a Chebyshev polynomial of k^{th} order, the output will be a sinusoid having harmonic frequency kf_0 [2, pp. 135-136]. A spectrum with multiple harmonics can be produced by combining polynomials of different orders.

Dynamic spectra can be obtained by making the distortion index α time dependent. The problem lies in the fact that particularly higher order transfer functions do not change the partial structure smoothly enough, and small changes in distortion index can cause

abrupt jumps in the spectral evolution. A possible solution is to alternate transfer function's even and odd term signs independently [2]. Inharmonic spectra can be synthesized by ring modulating the output of the waveshaper with another sound source.

Another problem arises because the distortion index is used to control not only the spectral content, but also the loudness of the sound. In principle, acoustic instruments have a tendency to sound brighter when they get louder in level, but it is rare to find a common modulation source for timbre and loudness. One possible solution is to control the output level with a separate scaling function, which is also a function of α [2].

Bézier Synthesis

Lang suggests a synthesis technique that uses cubic Bézier curves as source waveforms [18]. A Bézier curve passes through start point \mathbf{p}_0 and end point \mathbf{p}_3 with respect to two control points \mathbf{p}_1 and \mathbf{p}_2 , so that when t is varied from 0.0 to 1.0, following equations are satisfied [19].

$$\mathbf{p}(t) = \mathbf{a}t^3 + \mathbf{b}t^2 + \mathbf{c}t + \mathbf{p}_0 \quad (2.8)$$

$$\begin{cases} \mathbf{c} = 3(\mathbf{p}_1 - \mathbf{p}_0) \\ \mathbf{b} = 3(\mathbf{p}_2 - \mathbf{p}_1) - \mathbf{c} \\ \mathbf{a} = \mathbf{p}_3 - \mathbf{p}_0 - \mathbf{c} - \mathbf{b} \end{cases} \quad (2.9)$$

Control points are further constrained so that \mathbf{p}_1 shall always be within $[0,0] \dots [\lambda, y_{\max}]$ and \mathbf{p}_2 in the corresponding negative coordinate plane (Figure 2.4). Distance between start and end points defines the wavelength of the sound, while amplitude is determined by control points' relative distance to the x-axis. Control points can naturally be modulated either at control rate (changing harmonic structure of the waveform or overall amplitude smoothly), or at audio rate: Lang concludes that when a control point is modulated at y-direction, results are similar to those of AM, with additional spectral peak at modulating frequency. Modulating in x-direction produces an FM-like spectrum, again with additional spectral clusters centered at $2f_0$ and higher harmonics.

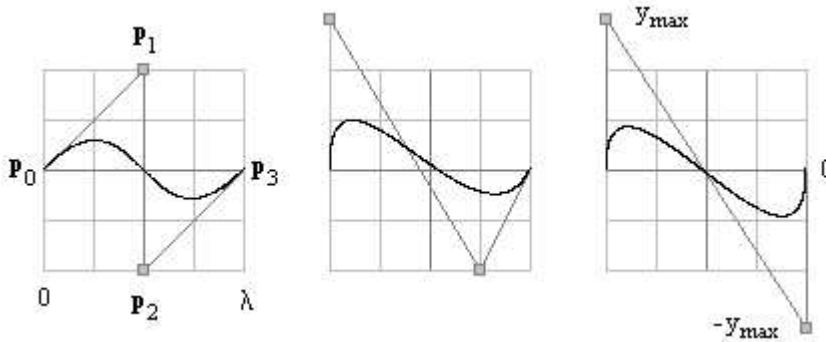


Figure 2.4 Some waveforms that can be produced using cubic Bézier curves.

2.1.5 Karplus-Strong (KS) Algorithm

A computationally efficient plucked string sound simulation was developed by Kevin Karplus and Alex Strong in early 1980's [20]. Figure 2.5 shows a diagram of a simple KS instrument consisting of a low-pass filtered white noise generator, delay line and a modifier. When a new note is initiated, the delay line is first filled with a burst of noise. The amplitude of the produced sound can be controlled by the amplitude of the inserted noise, and timbral variance can be produced by low-pass filtering the output of the noise generator before it is inserted into the delay line. Produced pitch is determined by delay line length combined with the modifier-introduced group delay.

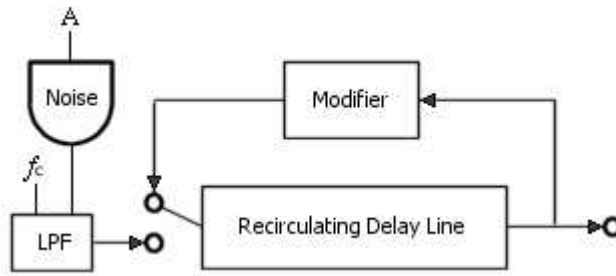


Figure 2.5 Simple plucked string implementation.

The actual sound producing stage of the synthesis process loops through the samples in the delay line, outputting and feeding each sample through the modifier back into the delay line for a next waveform cycle. If the modifier is omitted, a reed-like timbre is produced, having spectrum with all harmonics and equal amplitudes, and continuing to ad infinitum. In order to produce a naturally decaying amplitude curve in the output, the modifier can be constructed from a loss filter having less than unity gain, like the following simple first-order averaging low-pass filter:

$$y(n) = [x(n) + x(n-1)] / 2 \quad (2.10)$$

This generates sounds whose complex attack portion is quickly reduced to a sinusoidal form of oscillation. Overall decay time is inversely proportional to produced pitch, i.e. lower tones have longer decay time than higher ones, which is also the case with realworld acoustic instruments. It is possible to change the overall decay time by multiplying the right side of equation with a damping factor slightly less than 1. To stretch decay times, modifier can be written in form [1]

$$\begin{aligned} y(n) &= [x(n) + x(n-1)] / 2 \quad , \text{with probability } 1/s \\ y(n) &= x(n) \quad , \text{with probability } 1 - 1/s \end{aligned} \quad (2.11)$$

where s is the pseudo random stretch factor in range (0..1]. Other constructs for the modifier are possible, for example cheap percussion like timbres can be produced with the help of probabilistic blend factor [20].

The basic algorithm suffers from tuning problems, which is more profound at higher pitches (when length of the delay line decreases). In fact, some pitches in the upper part

of the keyboard range are off tune by more than one semitone. Jaffe and Smith proposed the use of interpolating filters in order to get a fractional delay line length [21], and introduced other important extensions to the basic implementation. They also studied the algorithm further, and realized that it was physically analogous to a sampling of the transversal wave on a string instrument, where feedback loop filter represents the total string losses over one period. Generalization of the algorithm lead to digital waveguide synthesis techniques. Sullivan applied the algorithm to electric guitar simulation with distortion and feedback [22].

2.1.6 Wavetable Synthesis

Wavetable synthesis techniques are based on a digital table-lookup oscillator, which generates its output by simply reading individual samples from an onboard memory wavetable, containing either pre-calculated or pre-recorded waveforms. Reasoning behind this approach is that a memory access requires much less computing resources than would be necessary for per-sample evaluation of a complex mathematical formula¹. Wavetable synthesis is very general, as any sound can be produced, but the downside is that it is rather inflexible in terms of real-time modifications [4]. It also requires a lot of memory when compared to other synthesis techniques, which can be reduced by looping, pitch shifting and data compression algorithms. It is a popular method however, due to its use in computer soundcards, mobile phones and dedicated sampler units.

The frequency of a digital table-lookup oscillator can be calculated from

$$f_0 = f_s \frac{SI}{N} \Leftrightarrow SI = N \frac{f_0}{f_s} \quad (2.12)$$

where SI is the sampling increment, f_s is the sampling rate and N is the wavetable size. To get different frequencies from same source material, SI must be changed, because in fixed sampling rate systems f_s is constant, and the same goes for N as wavetable contents are not usually modified. Changing SI effectively changes the size of wavetable either by skipping or holding samples during a table scan.

SI has usually a fractional part that must be converted to an integer index that is required by the table lookup procedure. There are several solutions, the simplest being truncating method which makes the translation by just brutally ignoring the fractional part. Rounding oscillators round floating point value to the nearest integer, which is more accurate, but excellent results can be achieved by using interpolating oscillators (linear or even higher order ones can be used). All of these translation methods add table

¹ These days, microprocessor cache handling scheme might result in a situation where shuffling a large wavetable between main memory and cache becomes the bottleneck, and simple calculations may actually be carried out faster than table accesses. In general however, it can be assumed that wavetable accesses are more efficient than brute calculations.

lookup noise to the outputted signal, and there is the traditional tradeoff between computational cost vs. error reduction.

Table 2.1 Signal to noise ratios for table lookup oscillators in respect to wavetable size k (in bits) [2].

method	SNR in dB	256 bytes	512 bytes	1024 bytes
truncate	$6(k-2)$	36	42	48
round	$6(k-1)$	42	48	54
linear interpolation	$12(k-1)$	84	96	108

Wavetable synthesis brand groups together a number of techniques that can be classified based on the number of wavetables that are *audible* at one time [1]. Multiple wavetables are usually utilized in all variations, as even small amounts of pitch shifting quickly destroy the character of original recording, and because different waveforms might need to be triggered in response to the intensity of articulation. In this light, Roads' term "multiple wavetable synthesis" is somewhat misleading, and perhaps it would be better to speak of "mixed wavetable synthesis" when the number of simultaneously audible wavetables exceeds one.

In the simplest form only one wavetable is audible, and the one that is picked from available waveset is based on triggering key's pitch and intensity (or velocity). This form is used in entry level samplers, computer soundcards and mobile phones, particularly those supporting Emu's SoundFont [23] or Downloadable Sounds [24] file format (latter is also part of MPEG-4 standard, known as SASBF [25]).

Mixed wavetable synthesis is a more advanced variation. In wavetable crossfading, two waves are mixed together so that while the first is fading out, the next one fades in. This can make a transition from a key or velocity zone to another sound somewhat smoother than in the single wavetable variation. It can also shape the sound of a single key press, as wavetables are faded in and out in time dimension, either automatically using amplitude envelope with initial delay stage, or via performance controlled slider or joystick. The third possible crossfading method is to use an interpolating oscillator so that initial waveform is gradually morphed into second wave.

Another mixed wavetable synthesis technique quite similar to crossfading is wavestacking, which is actually a form of additive synthesis. It uses complex waves as components instead of simple sinusoids. Any number of wavetables can be audible at one time, and each oscillator has its own dedicated amplitude envelope. These techniques have been implemented in more advanced sampler units, and were quite successfully used in late 80's synthesizers sharing the sampled attack - synthesized sustain paradigm. Vector synthesis machines belong also to this category.

2.1.7 Synergy of Synthesis Methods

This section attempts to find common components of different synthesis techniques presented in previous topics, in order to build a hybrid environment where various methods can coexist simultaneously. Such an environment would be capable of

producing a broad range of sound material, as strengths and weaknesses of individual methods can be taken into account.

It is obvious that an oscillator is employed by all methods. A table lookup method seems to be adequate for all other techniques except for noise generator (used in subtractive and plucked string implementations), and the Bézier oscillator. Wavetable technique allows any source waveform to be used, which can even be generated algorithmically from a spectral definition. Waveshaping just uses an indirection to get its output value from source oscillator material.

Modulation techniques call for oscillator interconnections, as do subtractive hard sync and additive mixing. A natural solution is to arrange oscillators into an audio rate modulation matrix together with feedback paths. Interconnection types would then include AM with variations, FM and PM, sync and simple arithmetic mix, with modulation amount set by modulating oscillator's level. Each oscillator defines also control rate modulation inputs for amplitude and frequency.

Filtering is exclusively used by subtractive method, as other techniques manage to shape spectra internally. Exception is the plucked string algorithm, which uses filters in excitation and feedback loops. However, as source material produced by any technique above can be spectrally rich, filtering might be proper in those contexts as well.

2.2 Effects

Dry synthetic sound often sounds synthetic. It can be characterized as being electronic, cold, boring and unnatural when compared to the sound generated by traditional acoustic instruments. On the other hand, it is not necessarily the physical construction of acoustic instruments that makes them more interesting to listen and play. It has also largely to do with the environment that the instrument is played in. Acoustic instruments radiate sound energy into three-dimensional space, and this space can add its own characteristics to the resultant sound, whereas electronic sound is outputted via more or less uni-directional loudspeaker system.

For this reason, from its early (but relatively late) beginnings, synthesized sound has been processed by effect devices, to make it more natural and more interesting to listen. At first, these devices were separate pieces of hardware, and were used to post-process the sound produced by a synthesizer in order to simulate a group of instruments playing together, or to mimic acoustic space with echo or reverberation units. The next step was to integrate outboard effects into the synthesizer unit itself, and when synthesizers were moving into the digital domain, also analog effects were replaced by digital circuitry. The post-processing approach prevailed, as effects were usually placed at the end of the audio processing chain, and at the time of this writing, that is the most common practice taken by many of a synthesizer. More importantly, effects can nowadays be considered as an integral part of the produced sound, as effect algorithm parameters are stored with the patch, and some parameters can be even modulated like conventional synthesis parameters. This makes them real sound modifiers, taking same kind of a role that filters

are usually seen in. Later synthesizer architectures also allow the amount of effect processing be separately adjusted for each subcomponent of a compound sound, in form of send / return busses (where a common effect is shared by all subcomponents, but with differing amounts), or as insert effects (where a dedicated effect is assigned to the subcomponent, with controllable dry / wet mix amount) .

Next sections describe briefly the effect types that are most commonly found inside a synthesizer, with a list of their most important parameters. It is customary that effect algorithms can be selected from a large set, but only a few of them can be active at a single time.

2.2.1 Time-based Ambience Effects

Delay repeats source material once or several times, simulating an echo reflecting from a distant surface. At the simplest form, only two parameters are used to control the algorithm: delay *time* (i.e. the time lapse between repetitions) and *feedback* to determine the number of repeats. More complicated delay units are also available, including stereophonic units with panning or cross channel feedback, multi-tapped models (where the entire delay line length is divided into number of shorter segments), ones that are tied to external song tempo, and units that simulate analog tape loop or bucket brigade implementations with a low pass filter in the feedback chain, or with slight randomness introduced to the delay time. Delays are most often used as send effects.

Reverb is used to simulate natural listening environments by introducing a pre-delay, early reflections (perception of room size) and decaying echo tails into source signal. Parameters can be used to specify reverb *type* (plate, spring, tape, room), *room size* or *decay time*, and modeled environment *texture* (high-frequency damping, diffusion, colour). It usually appears at the end of synthesizer's audio chain, and is often used as a send effect when working with mixing consoles.

2.2.2 Modulation Effects

When the length of a delay line is modulated with a periodic low-frequency signal, the pitch of the delayed signal is constantly altered and detuned from source signal. If delay times are kept small, the effect of multiple instruments playing in unison can be simulated, and sound produced can be characterized as thicker and warmer than the original source sound. **Chorus** effect is produced this way and is frequently found in synthesizers to fatten up single oscillator sound sources. Parameters include *mix* balance between dry and wet signals, *delay time*, *depth* of modulation (i.e. how much delay time is altered), modulation *speed* (how quickly delay time fluctuates between minimum and maximum values), and *width* determining stereophonic spread of the effect. Advanced units have multiple parallel lines for even thicker output, or selectable LFO waveforms. **Flanger** effect is similar to chorus, but uses shorter delay times and an additional parameter for *feedback*. **Vibrato** effect can be generated by setting *mix* parameter to 100 % wet. All aforementioned units are typically used as insert effects.

Amplitude modulation effects include **Tremolo** (achieved by modulating the source waveform's amplitude with an LFO set to a pulse or triangular waveform) and **Ring Modulation** where source signal's instantaneous amplitude is multiplied by another bipolar audio frequency signal.

Phaser effect can be created by mixing source signal with a delayed version that has been fed through a series of allpass filters. Each allpass stage (there are typically 4-12 of them) introduces a phase shift into the signal, and the entire string of filters can be tuned to produce a spectrum with non-uniform notch distribution. Characteristic dynamic sound of a phaser is then generated by sweeping notch positions up and down in frequency domain with an LFO.

2.2.3 Filters

Basic synthesizer filters (HPF, LPF, BPF, BRF) were already discussed in section 2.1.2. **Graphic EQ** is a special filter type, in which the whole audible frequency range is divided into a group of evenly spaced bands. Each band has a parameter that either boosts or cuts frequencies around the center of the band. **Parametric EQ** allows determination of a center frequency, into which the frequency boost or cut is applied. It usually has also a setting for the bandwidth around the center frequency that filtering operation affects. EQs are usually located at the end of the signal chain, and work as insert effects.

2.2.4 Dynamics and Gain Control

Compressor squeezes the dynamic range by attenuating high levels and boosting silent ones. Parameters consist of *threshold* (giving the level that input sound has to reach before compression becomes active), *ratio* between input and output signal defines amount of compression, and *attack* and *release* times in order to gradually fade the effect in and out. **Expander** has the opposite effect.

Limiter affects only high level signals by attenuating those that would bypass the set *threshold*. It is a harsher form of compression, originally used to maximize loudness without overloading the dynamic range. **Noise gates** were originally used to cut out background noise by setting a threshold value that a signal must reach before it is passed on to the audio path.

When dynamic range is overloaded, new high frequency components are introduced into the source signal. **Overdrive** was originally achieved by overloading valves of tube amplifiers, and is an essential effect when processing electric guitar sounds. **Distortion** or **Fuzz** is a more harsh sounding effect, and was originated when transistors were overloaded. Parameters include *drive* or *gain* amount (i.e. how much original signal is boosted beyond reference level).

2.2.5 Stereo and Panning

Automatic panner has an LFO or an EG attached in order to cyclically modulate stereophonic positioning of the source signal. There are also **psychoacoustic panners** that emulate the interaural differences and other distance cues in order to create 3D soundscapes through binaural channels, or to create pseudo-stereo signals from a monophonic source.

2.3 Control Signals

2.3.1 Internal Modulation

Without modulation generators, even the most sophisticated synthesis algorithm will sound static. The first group of control signals carries modulation signals that are pre-programmed to the patch, and are either applied automatically, or in response to an external performer event.

2.3.1.1 Envelope Generators (EGs)

The amplitude of natural sounds is not constant over time, but rather follows an envelope contour, characterizing a particular sound source. This contour can be split into stages, and each stage can be approximated by a straight line or exponential curve drawn between segment's start and ending points. A device that produces either piecewise linear or exponential functions of time is called an envelope generator. Traditional EGs have controls for attack, decay and release times, and one for sustain level (see Figure 2.6). Envelope generator is triggered with a key press: it starts from zero and goes to maximum amplitude in time set by attack time parameter, then fades to sustain level within decay time attribute, where it stays until the key is released. Finally, amplitude goes from sustain level to zero within release time parameter setting. Later designs feature complex multistage EGs with loopable sections.

The output of an EG is usually connected into a VCA module in order to produce changes in loudness levels, but it can also be used to modulate filter's cutoff frequency and oscillator's pitch. It should be noted that when EG is patched into an amplifier module, it should be unipolar, whereas in other cases there isn't such a restriction.

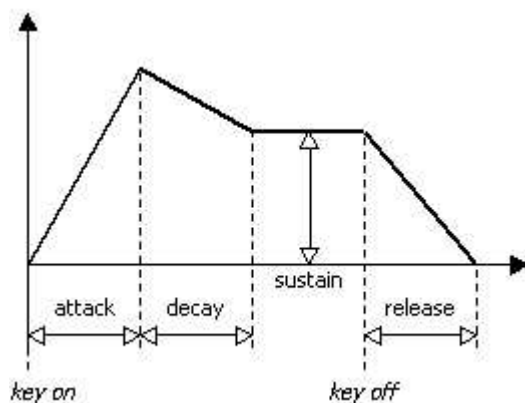


Figure 2.6 Traditional ADSR envelope.

2.3.1.2 Low Frequency Oscillators (LFOs)

A common performance technique is to introduce a slight vibrato into sound's sustaining stage. Such minute changes in frequency can be modelled by a low frequency oscillator operating on sub-audio frequencies, patched into OSC, VCA or VCF, which can result variety of effects ranging from subtle vibrato-like modulation to wide range frequency sweeps. Another frequency related routing is to vary audio oscillator's pulse waveform duty cycle width with an LFO.

Synthesis parameters of an LFO consist of waveform selection (where all familiar OSC shapes are available, including both up- and down ramping sawtooths and random waveform), frequency in range of 0.01..30 Hz, amplitude determining modulation depth, and delay for postponing vibrato to note's sustaining phase.

2.3.1.3 More Exotic Modulators

In contrast to continuous random value stream generated by random LFO waveform, the Random Value Generator (RVG) draws a new value only when explicitly triggered, for example in response to a note on event. Another triggered unit is Sample and Hold component, which samples current value in its input port whenever triggered, and keeps outputting that value until retriggered once again.

A Ramp Generator is like an envelope generator with a single attack stage : when triggered, its value goes to zero and rises back to full value at parametrized ramp rate. It is often used for fades and sweeps, and although it can be replaced with conventional EGs, its use is much simpler because there is only one parameter to control.

A Tracking generator is like a control rate waveshaper. It transforms a linear input value into output through a parametrized piecewise linear curve. As such, it can change the shape or scale of a modulation source, and is often used when creating custom key and velocity response curves.

A Lag processor smooths down sudden changes in control signals, allowing separate control over signal's rise and fall times. For example, when attached to oscillator's pitch, and triggered by a note on event, a portamento effect is achieved. There is no restriction on destination parameter, however, so it can process velocities and LFO shapes as well.

2.3.2 Performance Control

The second group of control signals transmits performance-related events (like instructions to play a note at specific pitch at a specific moment in time, with increased amount of vibrato) from physical controllers (like keyboard or modulation wheel) into synthesizer's control logic inputs. Alternatively, control signals may be generated by another machine. In analog domain that could be an arpeggiator, step sequencer or any other control voltage generator. With digitally controlled synthesizer, data could come from a computer application that has recorded musician's performance events, from an

edited musical score in electronic format, or even from an algorithmic virtual composer or accompanist.

With such a diversity of sources, it is important to have a universal interface specification to define how these devices can be made to understand each other. Analog control bus operates with control voltage levels, and in digital domain one of the earliest and still most widely used event stream format is MIDI, which defines both the electronic interface and the messaging protocol between musical instruments [26].

For the purpose, it defines a set of control sources and their expected destinations inside a 16 channel stream operating at the speed of 31.25 kbit/s. It is a serial protocol, where a byte consists of 10 bits (8 for data plus 2 for synchronization), so transmission time for single byte is 0.320 ms. Event packets are of one to three bytes in length (1 byte if running status is used), so the transmission of single event takes 0.320, 0.640 or 0.960 ms, typical case being the three byte version. Each controller event has a 7 or 14 bit data value associated with it, but unfortunately they are global to the whole channel, and the only note specific events are on-off gate, velocity and polyphonic aftertouch.

Table 2.2 MIDI channel voice messages. Numbers are hexadecimal, n in status byte encodes channel information. MSB = Most Significant Byte, LSB = Least Significant Byte.

Status	Event	Data 1	Data 2	
8n	Note off	note number	velocity	
9n	Note on	note number	velocity	
An	Poly pressure	note number	velocity	
Bn	Controller	control number	control value	
	"	00..1F	"	continuous controller MSBs (14 bits)
	"	20..3F	"	continuous controller LSBs (14 bits)
	"	40..45	"	switches
	"	46..5F	"	7-bit controllers
	"	60..63	"	(non)registered parameters
	"	64..79	"	undefined
Cn	Program change	program number	-	
Dn	Channel pressure	pressure value	-	
En	Pitch wheel	value LSB	value MSB	

2.3.3 Modulation Matrix

Modular synthesizers (see section 3.2) employ very flexible signal routing capabilities, as almost any source signal can be patched into any imaginable destination for modulation purposes. Semimodular synthesizers often route their modulation signals via a modulation matrix, which has a row (or slot) for each source-destination connection pair, column for each available destination, and a knob or parameter value at each junction to define the amount of modulation. Usually, single source can modulate any number of destinations, and each destination can have multiple sources.

The Downloadable Sounds standard takes the basic concept a bit further [24]. Connection blocks are used to define links between sources and destinations, and they consist of modulation source, control, scale, transform and destination. Sources are

either internal generators (LFOs and EGs) or external MIDI sources (such as key number, velocity or pitch wheel). A Control can be used to offset the modulation amount (e.g. mod wheel can influence to the amount of pitch change generated by an LFO source). A Scale value defines the preset value of destination or amount of modulation, and Transform is used to apply a specific mapping into destination's parameter space.

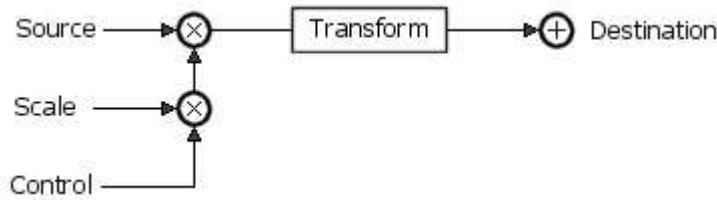


Figure 2.7 Connection Block as defined by Downloadable Sounds specification.

Yet another technique is used by Emu's desktop instruments [27], which have 36 virtual patch cords (i.e. slots) each with source, destination and modulation amount settings. They do offer similar flexibility that DLS specification achieves by allowing cord's modulation amount to be a possible destination of another cord.

Chapter

3 Sound Synthesis in Practice

Sound synthesizers are available both as hardware and software configurations. Due to flexibility issues, the latest trend seems to be moving towards software-based solutions, although hardware-based systems are not likely to disappear altogether. This is because their customized signal processing abilities still outperform those of general purpose PCs, and because they do not crash in actual performance situations.

Existing hardware implementations can be categorized into devices having either analog or digital sound production components, while software-based systems fall into batch-oriented or real-time environments. Structurally synthesizers can be grouped into hardwired and modular setups, based on scalability of their internal architecture. Hardwired designs are more performance oriented, enabling only limited freedom of interconnectivity between synthesizer elements.

3.1 Mixing Audio and Control Signals

Regardless of categorization, all sound synthesis systems face the task of combining the audio and control signals into a working whole. Analog synthesizers operate on voltage control principle, so mixing these signals is more or less a simple matter of connecting modulator output signal into the input of another module, including the necessary signal level matching.

In digital implementations signal busses are not physical entities, but rather their logical abstractions in forms of data structures. Buss contents cannot be mixed easily, because external control buss data must be parsed and converted from MIDI format into appropriate function calls and parameter values before it makes sense to feed it further to the synthesis engine. Even internal modulation data must be transformed and scaled so that the end result does not exceed system's dynamic range, and so that the audio rate aliasing is kept in minimum. This complicates processing requirements, and as a result, latency figures may become problematic and make the real-time use of a digital synthesizer impractical. This in turn impacts the synthesis algorithm complexity, making it necessary to employ synthesis techniques that can produce interesting material using a minimum amount of computing power and control data. Another approach is to reduce the amount of polyphony that the system can provide.

Roads calls the work that is involved with performance control data handling as event processing, and finds four subprograms to realize the task [1 pp. 698-701]. Processing starts when the control logic detects an input event and issues an interrupt request to the control processor, whose first task is to *parse* the incoming message according to MIDI grammar. Decoded event is then passed to *Voice Assignment* subprogram which converts event to DSP commands that should be executed in response to the input, assigns those commands to appropriate DSP voice channels and writes them to a list of required actions. *Event Scheduler* task is activated at each control rate tick. It checks whether action list contains an entry that should be executed at that point in time, sets DSP parameters accordingly and goes back to suspended state. Finally, *Resource Allocation* task is invoked when a note start or stop related event is encountered. It manages states of active voices and decides which voice is allocated for the new note, and in case of an overflow, replaces one of the active notes (using a fast amplitude envelope tailing ramp on old note) with new one. There are many strategies on how this dynamic voice allocation can be implemented, the most common being first, last, high and low note priority schemes.

Roads also discusses parameter update problem, which arises when control and internal modulation data is synchronized to the DSP task [1, pp. 950-955]. Parameter updates tend to occur at bursts and it might be difficult to keep up with audio rate processing if bursts take too long to handle. A fixed update period before each sample calculation cycle is one possible solution, and if there are too many parameters to update during a single period, pending updates are processed at the start of the next cycle. In worst case, this scheme might lead to an overflow of update queue, and might require selective discarding of parameter updates.

In addition to real-time data, the digital control buss is also used for routing more statically inclined patching data, containing snapshots (or presets) of the settings of all synthesis parameters. In such cases, sound output is usually turned temporarily off, and all currently playing sounds are retriggered after all parameters have been updated. This might be a desired behaviour anyway, since consecutive patches might produce so different sonic material, that without proper morphing there would just be annoying clicks in the sound output.

3.2 Hardware Implementations

Dedicated hardware synthesizers on the market are usually based on a fixed architecture model, and in general utilize only single synthesis technique in their sound production process. However, some recent models can mix several elementary synthesis methods together, but this is achieved by expanding the basic sound engine with addon boards, and there is no interaction between elements of different synthesis methods (besides simple arithmetic mixing of element outputs). Even if modulation between elements is possible, the number of oscillators is too small to gain any remarkable benefit out of it. Furthermore, the topology of the elements is essentially fixed.

Three high-end workstations from Roland, Yamaha and Korg were evaluated [28]-[31], and they all suffer from shortcomings described above. In their basic configuration, the units are wavetable-based with subtractive processing. The expansion options give access to analog modeling, physical modeling, and modulation synthesis techniques. Modulation EGs are relatively simple extended ADSR sources, and configurable modulation matrix is only available in Korg.

In addition to the large onboard sample pool, the strong points of the units are in the amount of polyphony, and in the versatile modifier and effects sections, although the latter is commonly located at the end of the audio chain.

Table 3.1 Summary of elements per voice in three hardware-based workstations. Letters in square brackets in EG row define stages [Attack, Decay, Sustain, Release, Initial delay, Hold].

	Roland V-Synth	Yamaha Motif	Korg Triton
Oscillators	2	4	2
Modifiers / Filters	2	4	2
Audio Modulators	1	-	-
Amplifiers	1	4	2
EGs	$13 = (4 + 2) \times 2 + 1$ [ADSR]	$12 = 3 \times 4$ [IAHDSR]	$6 = 3 \times 2$ [AHDSR]
LFOs	$5 = (1 + 1) \times 2 + 1$	$4 = 1 \times 4$	$4 = 2 \times 2$
Effects	3	$5 = 2 + 3$	$8 = 5 + 3$
max polyphony	24	62 + plugins	60 + plugins

3.3 Software-based Approach

Also software-based sound synthesis systems can be seen either as generic modular environments (which are realized with dedicated synthesis languages) or as fixed-architecture software synthesizer plugins (that are used within a virtual desktop studio environment). The latter is a more performance oriented system operating in real-time, while the implementations belonging to the former group are generally less impulsive offline processes, offering deeper level of interaction with the synthesis algorithms.

3.3.1 Synthesis Languages

Synthesis languages abstract the synthesis process with concepts like unit generators, instrument graphs, function tables, control structures, and messaging protocols between abstracted components [32]. They are extremely flexible, allowing even exploration of novel synthesis techniques, but at the same time suffer from this generality as the amount of parametrization data easily blurs the ultimate goal of music making. Batch-oriented nature with lack of proper performance control is another negative aspect of this approach.

Synthesis languages use either character-based or graphical interfaces. Examples of the former are MusicV based languages like Csound [33], and SuperCollider [32] which provides an object-oriented approach to sound synthesis. Graphical front shells have been developed for character-based languages as an afterthought, and as such they do

not compete with pure graphical languages like PD [34] and Reaktor [35], which bridges the gap between modular synthesizer design environments and real-time plugins.

3.3.2 Virtual Desktop Studio

Digital Audio Workstations (DAWs) of earlier days were giant hardware-based installations, using dedicated DSP devices hardwired into a custom topology, and were primarily intended for post-production audio editing chores. Each production site usually had their own proprietary systems, causing incompatibility problems and difficulties when adapting to different work scenarios. In smaller scale, synthesizer manufacturers discovered that extending mere audio synthesis capabilities of existing models with a few built-in effects and an on-board sequencer, a single hardware unit could serve as a self-contained recording studio, and decided also to call them DAWs. Extensibility issues remained still unsolved, as users were constrained to only those synthesis methods and effect processing algorithms that were implemented in the ASICs of the unit.

These days DAWs can be implemented in software, running on a standard personal computer equipped with nothing more exotic than a stock soundcard. These environments typically employ a desktop studio metaphor, where traditional recording studio consisting of synthesizers, effects, mixing desks and multitrack recording devices is exhaustively modeled in software. Outboard equipment is not necessarily required, but can naturally be patched in should processing load be distributed, or if their unique sound is preferred in a composition.

At the heart of every virtual desktop studio is a central *host* application which is the sole possessor of computer's AD/DA- and MIDI interfaces, and the provider of the master clock for various synchronization tasks. It contains facilities for routing real-time audio streams between *plugin* extensions, which are software modules acting as effect devices and software synthesizers, and storing those streams on hard disk for later manipulation. There can also be multiple hosts running concurrently, even over a LAN in another PC or as a dedicated hardware host unit, but there is always only one master host application multiplexing the resources of the soundcard. Desktop studio metaphor is further reinforced by the host with a virtual multitrack tape and virtual mixing desk MMIs, thus making musicians and studio engineers immediately familiar with the new concept. There are other metaphors used in DAWs, but that of mimicking the real world studio is by far the most common one.

3.3.3 Plugin Architectures

In summary, modern DAWs offer a versatile audio production system, which can be extended with 3rd party plugins, all working within the same basic hardware platform that is used for word processing and internet browsing. Each host manufacturer and operating system provider decided to implement a host-plugin architecture of its own, leading once again to incompatibility problems (Figure 3.1). SDKs for each are available for download via internet, however.










Architecture		Company	Defacto Host	Platforms	11 / 2004	05 / 2005	08 / 2005
	DirectX (instrument)	Microsoft Cakewalk	Sonar		115 (16 %)	134 (16 %)	139 (16 %)
	Audio Units	Apple	Logic		150 (20 %)	186 (22 %)	194 (22 %)
	Virtual Studio Technology	Steinberg	Cubase	   	468 (64 %)	532 (62 %)	557 (62 %)

Figure 3.1 Most popular plugin architectures. Last three columns show number of software synthesizer implementations per architecture in KVR Audio's database [36]. Data was sampled in november 2004, may 2005 and august 2005.

Plugin popularity was raised in 1996, when Steinberg released first version of their audio plugin format, entitled VST (Virtual Studio Technology), for 3rd party developers. It enabled development of audio processing plugins, i.e. virtual effect devices that were capable of real-time audio stream processing inside host environment. In 1999 it was time for version 2, which allowed creation of virtual instrument plugins, i.e. software synthesizers that were, again in real-time, capable of producing sound in response to MIDI messages streamed to them by a host application. As of this writing, current version is at 2.3.

VST is a cross-platform plugin architecture, supporting Windows, Macintosh and Unix operating systems. It contains also a VSTGUI extension, allowing platform independent MMI development. Latest SDK with examples and documentation can be downloaded from [37], consisting of an undocumented C-language interface, and of documented API in form of two C++ classes, one for each major SDK revision.

DirectX plugins are implemented as DirectShow filters, which themselves form a subset of Microsoft's DirectX API. They have an open and well-documented SDK from Cakewalk [38], but suffer from not being a cross-platform environment. Same arguments hold also for Apple's **Audio Units** [39], but because both of these architectures are closely linked to the operating system, they are supported by major host applications. On Linux platforms [40] host-plugin supply is far more minute, as major companies have ignored these markets altogether.

Cross-architecture use of plugins can be achieved by special adapters, which fit into the host's native plugin slot on one end, and to alien architecture on the other, making necessary protocol conversions in real-time. Especially VST-based plugs benefit from these wrappers, as there is a converter for all other major format.

3.3.4 Internal Structure of Plugins

Plugins have three real-time (possibly threaded) data streams to handle. Audio stream routes audio rate signals between host and plugin, while MIDI and Parameter streams transfer control rate signals relating to external performance data and internal synthesis

parameter automation. Following sections provide a comparison of AU, DXi and VST plugin architectures in these main areas of operation.

3.3.4.1 Audio Processing

All architectures investigated in this section are block-oriented, as it is more efficient to buffer the output data rather than calculate values on a single sample basis. Size of the output buffer depends on hardware and drivers that are installed on a particular computer, and on the amount of permitted latency. The dilemma is that lower latencies require smaller buffer sizes, but in order to produce drop-free audio output stream, a certain minimum size is required. All architectures support also multichannel audio, but in different ways, as discussed below.

AU plugins possess an extremely flexible audio processing architecture. Each plugin instance can have multiple busses, any number of input and output ports, and each in configurable sample data format. Port and stream configurations can even change dynamically. As a consequence, rendering method is given only timestamp, number of frames to process and a set of flags as parameters, and plugin is responsible for getting the input and output buffer pointers, as required by current patching setup.

DXi processing method is provided with timestamps for synchronization of parameter updates and MIDI events, single input buffer, configurable number of output buffers, and a queue of currently active MIDI events. All audio buffers organize multichannel data in interleaved manner, and both input and output sample formats are negotiated during initialization phase.

VST plugins always use 32-bit floats, and separate audio buffer is provided for each audio channel instead of interleaving. Number of audio ports is determined at initialization time and is fixed thereafter, which causes unnecessary CPU overhead as plugins cannot adapt to dynamic patching configurations (i.e. it is possible to feed monaural signal into a 2-channel input plugin, but plugin must process both input channels as it does not have any knowledge of current patching topology). This can be avoided by compiling separate versions of single plugin for different port configurations, but it is obviously not an ideal solution.

VST plugin has two virtual methods which the host calls periodically in order to route audio data between itself and the plugin. *process()* is typically called when the plugin is patched as an aux buss send/return effect, and desired action is to accumulate effect's output into the material already present in the buss. *processReplacing()* is a faster method, as plugin can write over anything that is currently in the output buffer, and is used for insert effects and virtual instrument plugins. Parameters for both methods are equal, and prototype for *process()* is

```
virtual void AudioEffect::process(float** inputs, float** outputs, long sampleFrames);
```

where *sampleFrames* gives the number of samples that the plugin should process in response to this call, and is usually same value that was notified for buffer size during initialization time (it is not necessarily equal, but it is guaranteed not to exceed that

value). *inputs* is an array of 32-bit floating point audio streams, originating from a recorded audio track, live soundcard input or another plugin. *outputs* contains an array of output streams, and is naturally the place where plugin should store its calculated samples. These buffers may overlap, which should be taken into account when designing the processing algorithm. Samples are scaled into the range of $[-1..1]$, inclusively.

3.3.4.2 MIDI Event Processing

All architectures are capable of streaming timestamped events to plugins, within single sample accuracy. Like audio data, MIDI events can come from pre-recorded MIDI track, live from soundcard's MIDI input, or from another plugin or application. There are differences in amount of preprocessing, however.

AU plugins get their voice and sysex messages in form of parsed callbacks, based on event's status byte. Events are not structured, and raw MIDI data is passed as parameters of the callbacks. Dispatcher can also be hooked at an earlier stage, for example to perform channel based filtering, but then without the parsing support. An important extension is the possibility to use fractional data values for notes and controllers (in contrast to MIDI's 7 and 14 bit integer ranges), and the ability to have multiple instances of same note inside a single channel.

In **Dxi**, each input event is packed into an event structure, including a timestamp and actual data in event type specific format. Voice data is included in both raw and parsed form, and sysex data as a separate buffer. There can also be meta events encoded, and single event may group several raw MIDI events into single logical instance (e.g. note on-off pairs). This event is then sent to plugin for global filtering, and inserted into a pending event queue if it passed the filter. During rendering, pending events are first timestamped in relation to the current audio block, and those fitting the current timeslice are routed through plugin for extra initialization, and moved into a queue of active events, which is then given to the actual DSP processing callback. Finally, the plugin has a chance to release any extra memory allocated for the event before it is discarded.

VST events are also structured, including a timestamp, raw MIDI data, and some note-related parameters. Sysex data is not transferred¹, but future versions may provide also variable-length MIDI messages, as well as raw audio and video related events. It is also possible to generate MIDI events from plugin. There can be only one MIDI input port per plugin, which is usually opened when plugin is powered on from host's virtual instrument rack. While opened, events keep coming in until plugin's event processing

¹ This limitation means that standalone patch editors or hardware programmers cannot be used in conjunction with virtual instruments. This is a pity especially when emulating a particular hardware synthesizer, considering the variety of such tools already available. It is possible to overcome the limitation by using files for data transfer, but obviously then one loses the ability to manipulate parameter data in realtime.

method returns a value indicating that it does not want them anymore (see Figure 5.3), or until plugin is switched into suspended state. Prototype for the virtual MIDI input processing method is

```
virtual long AudioEffectX::processEvents(VstEvents* pEvents);
```

where *pEvents* contains an array of *VstMidiEvent* structures (each holding data for a single MIDI event), and an item indicating the number of elements in that array. For note events, extra information of total note duration, offset from start of the note, note off velocity and microtuning is included. It is important to note that VST does not provide any kind of higher level event support, so plugin has to start its event handling mission from parsing of raw MIDI data. Event processing also takes place in a thread of its own, so data received should be copied into plugin's internal storage, as *processEvents()* may be called multiple times per single audio processing block. Finally, SDK does not guarantee that events are sorted according to timestamp, which might be considered as a design fault.

In summary, event scheduling is provided only by the DXi framework, and voice assignment and resource allocation are left to plugin's responsibility in all architectures. VST provides least support, as even message decoding must be handled inside the plugin code.

3.3.4.3 Parameter Handling

Each AU plugin parameter has an index, label, unit, range, and default and current value attached. Host queries information of parameters during initialization time, and uses that data to tie MMI controls into plugin's parameter space, and to provide parameter automation facilities.

DXi parameters have an additional internal range mapping into DSP parameter space, an automation curve definition (for linear interpolation tasks) and a list of associated enumeration values, if applicable. Current and interpolated automation values are accessible within processing method using a single method call.

VST parameters are indexed and associated with label, unit label and current value that is always of type float and scaled into range [0..1]. Default MMI can display value as a plugin provided string, but like labels, the maximum length of that string is constrained to 8 characters.

VST plugins can update their synthesis parameters either from MIDI input port by responding to various control change messages (see previous section), or by receiving a stream of parameter automation data originating from host's automation tracks. These automation events can usually be drawn as multisegment curves using host-supported tools, but they can also originate from plugin's MMI and recorded into host's automation track.

```
virtual void AudioEffect::setParameterAutomated(long index, float value);  
virtual void AudioEffect::setParameter(long index, float value);
```

First method is invoked by plugin when user changes one of the automated parameters in the MMI so that the host is able to record parameter changes, if so desired. The second one is callbacked by host when such a stream is played back in order to update synthesis parameter and MMI controls accordingly.

3.3.5 Example Plugin Instrument

As mentioned earlier, most plugin instruments have fixed internal architecture, perhaps with configurable modulation structure at the most. One notable exception is VirSyn TERA [41], which is a modular plugin with six prewired setups employing common components of TERA's modular synthesis engine.

Source section consists of six oscillators. There are three single cycle OSCs with supersaw, FM, sync and waveshape mutation, white and pink noise sources and a spectrum oscillator. *Modifier* section has a waveshaper, ring modulator, two multimode filters, formant filter and a wave delay module for physical modelling tasks. *Mixer* section has five inputs, two submix outputs, and two total outputs (one equipped with a fixed lowpass filtering). *Output* section is at the end of the audio chain, and consists of amplifier and three insert effects.

Control rate modulation setup is quite exhausting, with four DADSR and four multisegment EGs, and four LFOs. These sources augmented with MIDI controllers are routed via 20-slot modulation matrix into any of the 81 destinations available. There are also four vector control surfaces allowing control of up to 64 destination parameters simultaneously, and a 64-step sequencer with multiple patterns to serve as a time-based modulation source.

The modular architecture of TERA makes it clearly more flexible than hardware workstations. In fact, TERA is capable of realizing at least in principle all synthesis methods that were described in section 2.1. The choice of effects is considerably narrower, but as plugins are intended to be used in conjunction with other effect plugins, it isn't really a problem.

Table 3.2 Elements in VirSyn TERA.

Oscillators	6
Modifiers / Filters	6 = 3 + 3
Mixers	2
Amplifiers	1
EGs	8 = 4 + 4 [DADSR + MEGs]
LFOs	4
Effects	5 = 3 + 2
max polyphony	64

Chapter

4 Requirements Specification

The design phase of PHUZOR is conducted using a software engineering standard PSS-05-0 developed by the European Space Association [42], in conjunction with object-oriented methodologies [43]. The standard approaches software development project by forming a lifecycle model of the software product, and finds six prominent milestones in the process. As the scope of this thesis is restricted to building a working prototype, only four of the phases are accomplished (i.e. acceptance testing and maintenance phases are not carried out).

The first of the accomplished phases is the user requirements capture, which in this thesis is combined with the software requirements definition phase, and described in this chapter. Architectural design phase follows in chapter 5, and the implementation issues are discussed in chapter 6. In short, requirement specification phases describe what needs to be done, architectural phase designs how it is done, and the implementation details the work in algorithmic level. Object-oriented design techniques utilized include use cases and object models in several levels of detail, using in UML notation [43].

Section 4.1 states general requirements of PHUZOR, and sections 4.2 through 4.4 give detailed model description of a general software synthesizer that should be applicable to any synthesizer architecture. As an extension of this general model, the synthesis architecture description of PHUZOR is given in section 4.5. Finally, section 4.6 describes various specific requirements, in particular the MMI in 4.6.3. The complete requirements specification is included in [48].

4.1 General Description

PHUZOR shall be a real-time audio synthesizer with polyphonic sound generation and dynamic synthesis parameter manipulation capabilities. It shall be implemented entirely in software as a VST plugin, and run in a standard personal computer equipped with off-the-shelf audio card, or mainboard integrated audio chipset. Note articulation commands and other controller data shall be generated by external MIDI devices, so dedicated hardware is not necessary.

A variety of synthesis techniques shall be utilized in order to produce a wide spectrum of sound material. Although common parameter spaces shall be used as much as

possible to control these diverse synthesis algorithms, relatively large amount of control data is needed to allow expressive performance. To make musician's interface simpler, a parameter grouping feature shall be implemented so that single controller movement can have effect on a number of individual synthesis parameters. These macros could affect fundamental timbral elements such as brightness or vibrato.

PHUZOR shall also be equipped with basic sound modifiers in form of patchable effects having dynamically controlled parameters. This is because the use of effects should be considered as an integral part of produced sound, and not as an external make-up that is applied to disguise shortcomings of a particular synthesis method. In prototype, effects shall be simpler than in the final product, and shall be included for architectural completeness in mind. They can always be bypassed altogether (to hear dry sound only, or to route them via host's dedicated effect plugins for more professional sound quality).

Table 4.1 PHUZOR general specifications

type	synthesizer plugin
polyphony	configurable, limited by host CPU resources
multitimbral	no
IO	audio out (2), audio in (2), midi in (1)
patching	semimodular
patch memory	1 active
patch database	browser, number of patches limited by host HDD (stored as files), subpatches
Waveform memory	limited by host RAM, stored as files
Synthesis	
type	hybrid : wavecycle, sample playback, subtractive, (group) additive, waveshaping, analog modelling, plucked string
oscillators per voice	1..8, max 56 (using 7 pseudo-oscillators in each of the 8 main oscs)
Modulation	
audio rate	FM, AM + RM, logical, sync
control rate	64 EGs, 64 LFOs, 32 RVGs
modulation matrix	128 slots (40 prewired + 88 patch), 320 sources, 448 destinations
controllers	all external midi controllers assignable via modulation matrix, macros for parameter grouping
MMI widgets	32
Effects	
number of FX units	2 inserts / oscillator, 2 inserts / particle, 8 global (4 inserts + 4 aux)
number of FX types	8

4.2 System Context

Boundaries between PHUZOR and its environment are shown in Figure 4.1. Actors are drawn using solid boxes, while data streams are surrounded by dashed rectangles. Audio signals are presented as outlined arrows, and synthesis control and parameter related data as dashed lines. Basic plugin control (such as instantiation and mode changes), and MMI-related information channels are indicated by solid black lines.

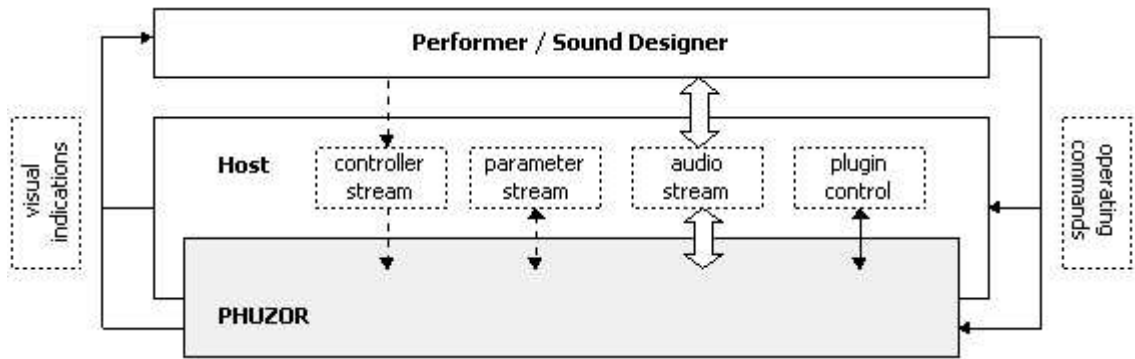


Figure 4.1 Context Diagram in form of top level data flow diagram.

The person using the system can take two interwoven roles. As a performer, he selects pre-programmed patches and plays PHUZOR in real-time as a musical instrument, while simultaneously manipulating synthesis parameters that affect the produced sound. As a sound designer, he creates and modifies preset patches and stores them for later use. PHUZOR shall be implemented in such a way, that these two roles are virtually inseparable, and for this reason, Performer is used in this thesis to refer the user of the instrument, regardless of his role. For Host, controller, parameter and audio stream descriptions, see section 3.3.2.

4.3 Use Cases

This section summarizes how Performer and Host target PHUZOR at the topmost level. Space limits restricts presentation here to a single example case and to a summary table listing all specified scenarios. The complete use cases are in [48].

UC-12	Performing with External Controller
Summary	PHUZOR responds to performance input by synthesizing audio
Frequency	By request. Commands take 0.320 .. 0.960 ms to transmit.
Usability reqs	Latency time shall be less than 20 ms.
Actors	Performer, Host
Preconditions	1. Input command is recognized [Unrecognized command] 2. Voices are available [Voice stealing]
Description	1. Performer interacts with external controller, or 2. Host plays back controller stream 3. See postconditions
Related cases	UC-10 : Configuring PHUZOR Functionality UC-12 : Performing Using External Audio Source
Exceptions	Unrecognized command : Ignored Voice stealing : One of the active voices is replaced, or command is ignored
Postconditions	PHUZOR interprets commands and generates sound according to parameter settings in current patch.
Notes	<i>Voice stealing algorithm can be selected as part of preferences { UC-10 }. There shall also be provision to gracefully stop all sounding voices without damaging ears or speakers. Simultaneous audio stream is possible, see { UC-12 }.</i>

Figure 4.2 Example use case describing actions involved when performer plays the instrument.

Most of the use cases are linked to the MMI tasks, as they are triggered by Performer's actions. In particular, boundary conditions and exceptions are of great importance when designing the dynamic behaviour of the system, as are the usability requirements giving quantitative performance related indications. They are summarized separately in section 4.6.4.

Table 4.2 Use case summary.

#	Description	Actors	Type	Related
1	Selecting a Patch	P	Patch	2,3
2	Storing a Patch	P	Patch	
3	Browsing a Patch Collection	P	Patch	1,10
4	Viewing Patch Parameters	PH	Patch	
5	Editing Patch Parameters	PH	Patch	8
6	Initializing Patch Parameters	P	Patch	2,10
7	Storing a Patch as Default	P	Patch	2
8	Defining Midi Controller for Editing a Parameter	P	Patch	9
9	Disconnecting Midi Controller from Parameter Editing	P	Patch	8
10	Configuring PHUZOR Functionality	P	Plug	
11	Performing Using External Controller	PH	Perform	10,12
12	Performing Using External Audio Source	PH	Perform	11
13	Changing Operating Mode	PH	Plug	
14	Loading and Initializing PHUZOR	PH	Plug	6,15
15	Unloading PHUZOR	PH	Plug	
16	Opening and Closing Main Window	P	Plug	2

4.4 Conceptual Object Model

4.4.1 Conceptual Class Model

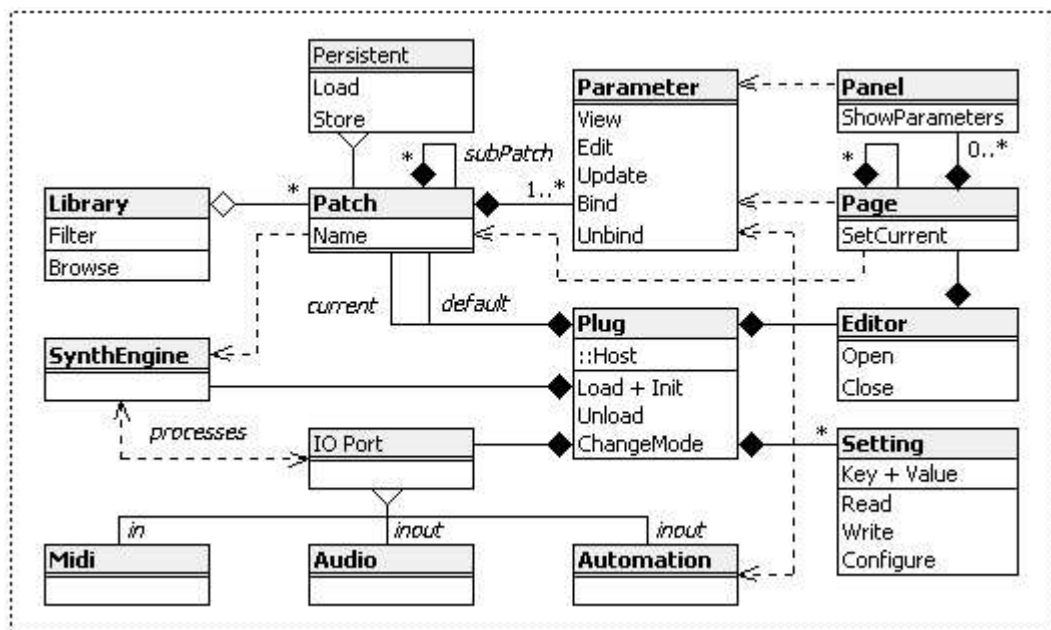


Figure 4.3 Conceptual Class Diagram.

Figure 4.3 shows the specification level object model of PHUZOR using UML notation (entire model dictionary is in [48]). The Host streams of Figure 4.1 are represented as three ports that are inherited from an abstract IO Port class, and both MMI-related streams are interfaced by the Editor class encapsulating the main window. In addition, MMI contains dedicated pages for further synthesis parameter manipulation tasks, and these are modeled above using an abstract class Page, which can further hold a number of subwindows abstracted by Panels.

Pages and Panels provide views to the current patch and to the parameters inside that patch, so a link is drawn from Page to classes Patch and Parameter. Parameter is an abstract class encapsulating single synthesis parameter, and a collection of these is held in a Patch, which defines single synthesizer timbre. A subPatch link groups functionally related parameters together, and allows consequently separate archival of units, as Patch is derived from Persistent class offering interface to disk files. Library is a collection of patches, and shall be used in future version to handle patch management related tasks.

SynthEngine encapsulates DSP specific functionality of a software synthesizer. It uses Parameters of the current Patch in order to produce distinct timbres, and is responsible for real-time MIDI and Audio stream handling. Automated synthesis parameters are updated via a direct link to Parameter class. Finally, to keep all classes together, model includes a Plug class that is responsible of core tasks associated with the Host environment interfacing.

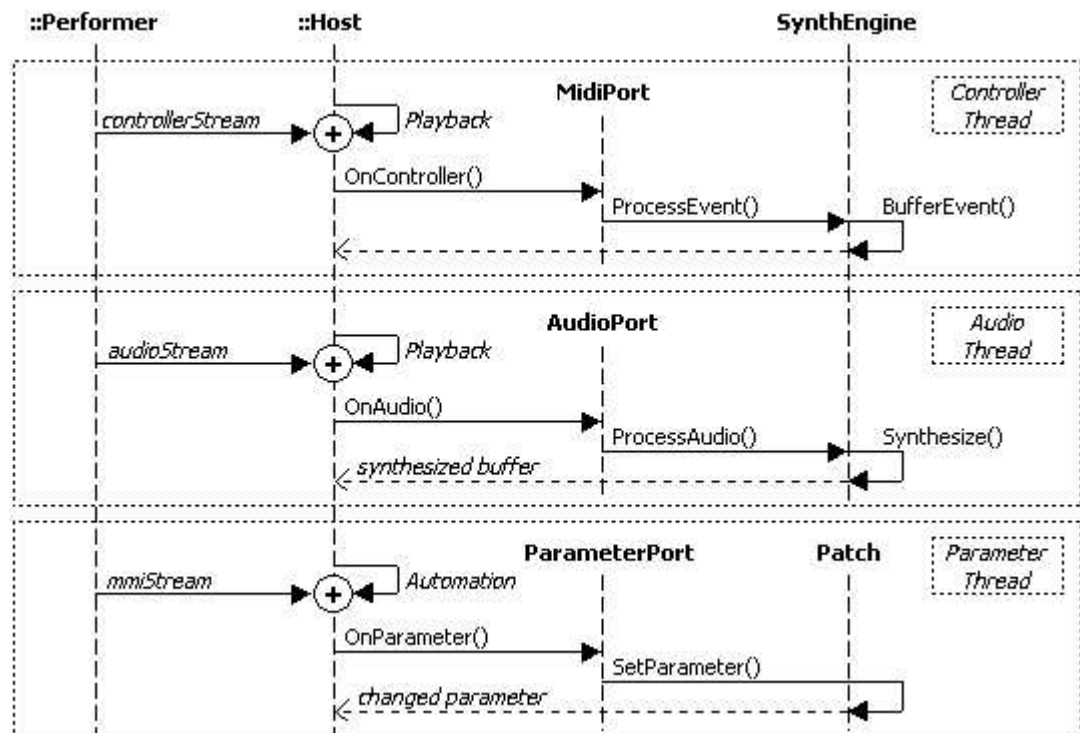


Figure 4.4. Sequence diagram for performing scenarios (UC-11 and UC-12).

4.4.2 Dynamic Model

The example use case of Figure 4.2 is drawn as a sequence diagram in Figure 4.4, describing the temporal relationships of actions that relate to performance tasks. Performer generated real-time streams shall be mixed with the streams provided by the Host's playback system, which are then routed to the appropriate input ports. They shall handle input streams concurrently, i.e. there are three simultaneous threads running in host context.

Timestamped controller stream shall be buffered for later use in audio thread, which synthesizes the audio stream according to buffered events, and returns that stream back to the host for further processing and output. Synthesis engine is not burdened with the parameter stream, as it is routed directly into / from the current patch memory. Synchronization to host tempo shall be handled inside audio stream.

4.5 Synthesis Architecture and Parameters

The most important design decision with any hard-wired synthesizer (in contrast to fully modular setup) is the definition of its synthesis architecture, i.e. the selection of individual synthesis components, their interconnections, parameters, and modulation routings between sources and destinations. Complete list of synthesis parameters is in [48], and the architectural structure is presented in Figure 4.5 of next page.

There are four sections in Figure 4.5. A voice is a collection of synthesis elements that are triggered by a single note event, and in PHUZOR that shall be equal to Source section. Subelements of Source (P1..P8) share synthesis parameters across all active voices, and the sum of these can be manipulated inside the Line Mixer section (e.g. P1 outputs of all active voices are summed together, and handled in C1). To complete entire patch definition, global Master Mixer and Modulation sections shall be included.

Audio signals shall be generated inside Source section, and shall then be routed through Line and Master Mixer sections into audio outputs. Modulation section shall generate control signals that can drive parameters of all sections, including itself. In addition to control rate modulation, Source section can route audio signals internally so that audio rate modulation is also possible.

4.5.1 Source Section

The Source section shall consist of eight *Particles* (P1..P8) and of an external audio source (EXT). The output of each particle can be routed to the Line Mixer section when it's generating audible waveform data, and/or to modulation input of another higher order particle when it is acting as an audio rate modulator. Single particle output can modulate up to seven destination particles, which in turn can have as many as eight modulating particles (when EXT is counted in) active at the same time.

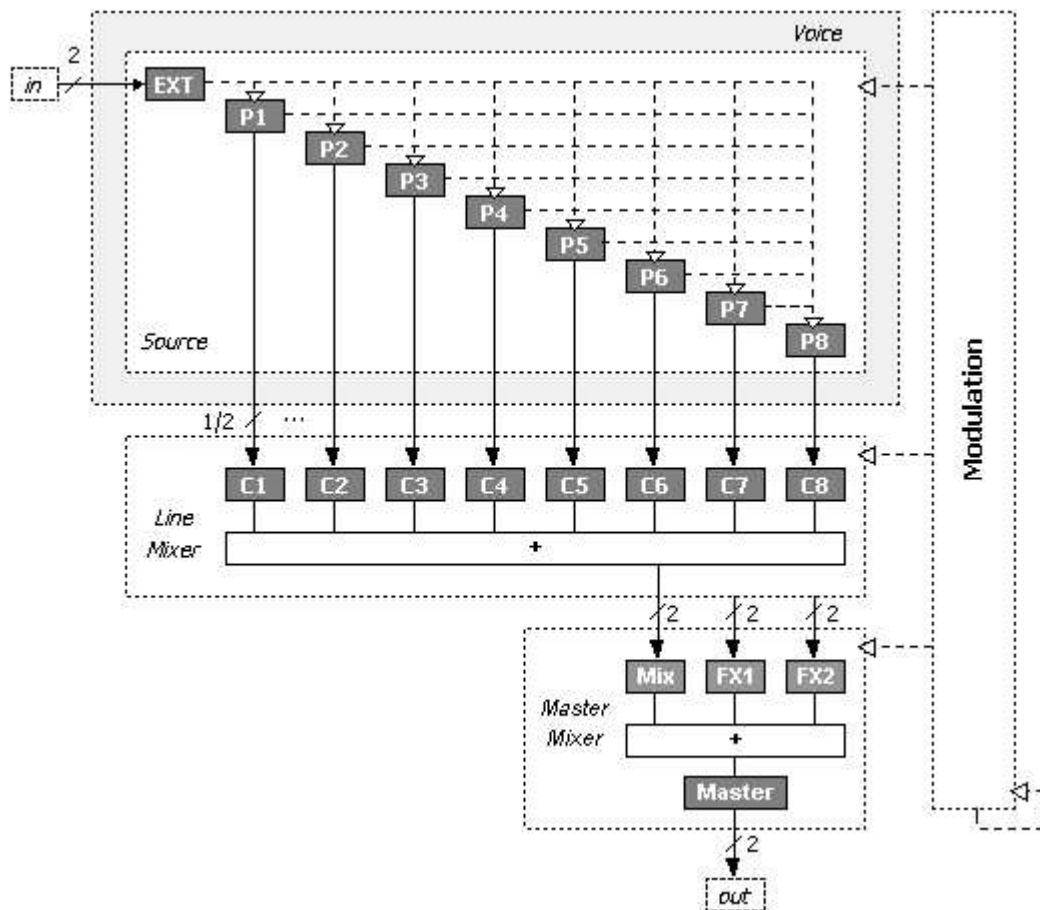


Figure 4.5 Synthesis architecture of PHUZOR. Solid lines denote audible signals, and dashed lines audio and control rate modulation signals.

Each particle shall have audio rate amplitude and frequency modulation inputs, but only one type of modulating algorithm can be active at an instance. For example, for a single input particle, it is *impossible* to have some modulators performing RM while others are doing FM, nor is it possible to mix two types of amplitude modulation algorithms together. This limitation shall only affect single particle though, as there can be RM on particle P2, and FM on P3. There shall also be other particle intermodulation options beside amplitude and frequency modulation, and these are listed below.

Table 4.3 Particle's audio rate modulation algorithms.

Algorithm	Operation	Description
FM	$\text{OscB}(\text{OscA})$	Frequency modulation
AM, RM	$\text{OscA} * \text{OscB}$	Amplitude modulation (multiplication)
XOR, OR, AND	$\text{OscA} \wedge \& \text{OscB}$	Amplitude modulation (logical)
Mix	$\text{OscA} + \text{OscB}$	Addition
Sync	$\text{OscA} \rightarrow \text{OscB}$	Slave oscillator synchronized to master
Pass	$\text{OscB} = \text{OscA}$	Replacement

In order to be able to modulate EXT input, a special intermodulation type 'Pass' shall be included, which disconnects internal oscillator of a particle and replaces it with input that is fed into audio rate modulation input. Each particle can also perform self modulation. The amount of modulation shall be determined by the product of modulator's output level and destination particle's input level settings.

Using this kind of patchable source section topology, PHUZOR shall provide freedom of modular systems, and allow wide variety of synthesizer architectures to be imitated. For example all original Yamaha DX7 [49] algorithms can be realized (not including carrier-to-modulator feedback path of algorithm 6). Matrix setups shall be static in nature, and can be stored and loaded independently of other patch parameters.

4.5.1.1 Particle

Figure 4.6 displays the internal structure of a particle. As can be seen, there shall be three particle algorithms, each utilizing a different synthesis technique. Class A shall be shared with wavecycle and sample based oscillators, the difference is implied by oscillator's waveform selection (sample data can either define a single cycle or a multicycle waveform). Modifier is discussed later in this chapter, but it shall usually contain at least one filter component. The amplifier component shall be included for a modulatable output and feedback level parameter, and it shall be identical in all particle classes. Class B shall use a waveshaper instead of a modifier, and shall contain some unique parameters, so it was decided to create a dedicated class for waveshaping alone (it could be also implemented by using a waveshaper modifier in class A particle). Finally, class C particles shall utilize plucked string algorithm, which is notably different from previous classes. In future versions class C shall be used as a general driver - resonator system, enabling more advanced physical modelling capabilities.

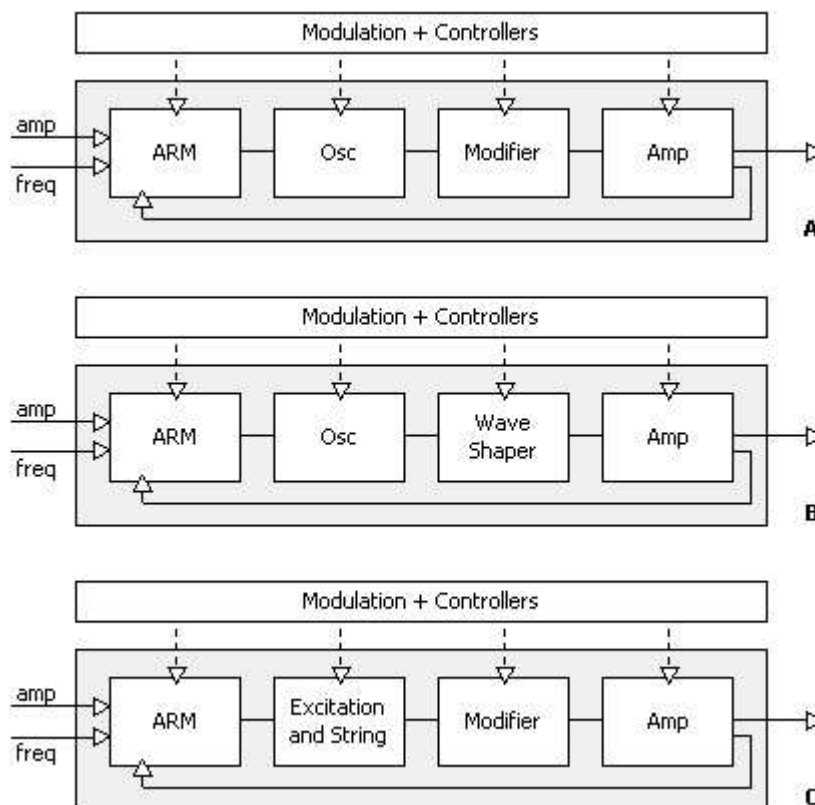


Figure 4.6. Particle classes. A) Wavetable B) Waveshaper C) Plucked String. Single particle is actually a self-contained synthesizer unit in itself. ARM = Audio Rate Modulation.

All internal particles (P1..P8) shall be monophonic. If the Line Mixer channel of the particle is in stereophonic mode, the output of the particle can be panned into stereo space, but in essence the sound shall be monophonic. There is an exception to the rule, however, as when sampled waveform with two channels is selected as an oscillator waveform, both channels shall be output to the Line Mixer. EXT particle can also feed stereophonic output, and when it is routed to one of the internal particles, choice of channel shall be made available.

In summary, fundamentally different synthesis techniques shall be implemented as unique particle classes, which are interchangeable inside the source section topology. Modulation synthesis techniques such as FM shall not be dedicated particle classes, but shall be realized as audio rate modulation interconnections between particles. At least the following techniques are realizable (and even within a single patch).

Table 4.4. Realizable synthesis techniques in PHUZOR.

Technique	Recipe
Additive	
sinusoidal	up to 8 partials, with dedicated frequency and amplitude envelopes and initial phases
group	like sinusoidal, but using offline generated single cycle wavetables instead of single sinusoid
Subtractive	
analog emulation	single cycle waveforms + filter + amplifier, multiphase oscillators, hard sync
sample playback	multicycle waveforms + filter + amplifier
Wavetable based	
crossfading	attack waveform in one particle, looped sustain waveform in another, mixed in time
wavestacking	like additive, but uses sampled waveforms
Waveshaping	
Chebyshev	by using dedicated particle
Bezier	by using dedicated particle
general	by using wavetable particle with waveshaper modifier
phase distortion	Casio CZ style can be emulated with general method described above
Physical modelling	
Plucked string	(extended Karplus-Strong algorithm) by using dedicated particle
Modulation	
FM	by audio rate modulation
AM + RM	by audio rate modulation
logical	by audio rate modulation

Audio Rate Modulation Input Block (ARM)

ARM input block shall define audio rate modulation topology between particles. *Active* can be used to quickly toggle modulation input port on and off, which might be useful when programming patches, but it is more CPU effective to cut the modulation connections altogether. *Source* particle list shall define those particles that act as modulators to the particle containing the ARM block. *Modulation type* shall be used to define intermodulation algorithm, and input *level* shall define amount of modulation. It shall be a control rate modulation destination, in order to synthesize dynamic spectra.

Oscillator Block (DCO)

DCO parameters shall be particle class specific, but shall contain also a common set that is shared by wavetable and waveshaper particles, including the usual *waveform* (sine, triangle, square, pulse, sawtooth, noise stock waves, and samples), and *pitch* related parameters. Of special interest are the fatness parameters available for the wavetable particle, which shall work with multiphase pseudo oscillators. When *fatness spread* is set to a value other than zero, the internal oscillator shall be surrounded by pseudo oscillators, one group detuned to higher and the other to lower pitch than the internal oscillator. *Fatness detune* shall define the number of cents that pseudo oscillators are tuned away from the center (either as positive or negative amount). *Fatness level* shall define level of pseudo oscillators in relative to internal DCO. This simple algorithm shall allow output of up to 7 detuned oscillators from single particle, and will fatten its sound quite a bit. When applied to the sawtooth waveform, the characteristic supersaw sound of Roland synthesizers can be produced.

It should be noted that contrary to common approach taken in literature, DCOs in PHUZOR shall always output signals with maximum amplitude, because control rate amplitude modulation input is moved into the DCA block. Wavetable particles with single cycle waveform shall have a parameter defining the *phase* of the waveform, and if waveform is set to stock pulse, *pulse width* parameter shall be available. Both of these can be control rate modulation destinations.

Multicycle particles shall allow more control over playback settings, like parameters for *looping* and sample *start offset*. In case of a multisample spread to cover entire keyboard range, extra settings provided shall be shared by all individual samples. This is a limitation of prototyped version, and the final release shall allow settings to be made to each sample separately. This shall be valid also for DCO common pitch parameters.

Amplifier Block (DCA)

This common block shall contain two amplitude-related parameters, both of which shall be modulation destinations. *Level* shall control maximum output amplitude of the particle, and *feedback level* the amount of signal that is sent back to particle's modulation inputs. Self-modulation destination shall be currently active modulation type.

Waveshaper Block (DCW)

Modifier block shall be replaced by a DCW block in the waveshaper particle. It shall be more advanced than a simple waveshaper modifier, as it shall have a dedicated Chebyshev *mode*, which can be used to produce any harmonic spectrum with direct definition of harmonic amplitudes. In Bézier mode, oscillator waveform shall be defined using a spline with two control points in two-dimensional plane.

Plucked String Block

The oscillator block of a plucked string particle shall be realized using an excitation waveform that is loaded into a virtual string in order to describe its initial displacement positions. All pitch and waveform related common DCO parameters shall be available, with additional parameters defined by extended Karplus-Strong models, including attack and pick related parameters (*strength*, *attack length*, *count*, *pick position* and *material*), *decay*, string *stiffness* and feedback parameters (*gain* and *pitch*). Note that amp distortion can be modelled using a waveshaper modifier inside the particle and that feedback shall be formed by particle self-modulation.

4.5.1.2 Modifier Block (MFX)

Modifier block shall be a particle component, but can also be located at three other places in the signal chain (referring to Figure 4.5, modifier block can be found inside any component with dark gray color, excluding the EXT box). It shall comprise two modifiers A and B, which can be connected in serial or parallel fashion. It shall take an audio source signal, apply selected processing algorithm to the signal (with real-time modulation inputs), and output the resulting audio stream. Audio input and output ports shall be either mono- or stereophonic depending on location of the modifier block. Modifiers in PHUZOR shall include filters, waveshapers and dedicated effect units.

Modifier blocks shall perform different tasks depending on their location. The block inside a particle shall facilitate oscillator-specific insert processing, and there can be a maximum of 8 of these blocks in simultaneous use for each voice. Depending on the Line Mixer channel settings, it shall have 1 or 2 outputs, but usually only one input (2 inputs with stereophonic sample playback), and shall consume more CPU power than the modifier blocks in other locations. Modifier block inside a Line Mixer channel shall be an alternative place to perform insert processing, and depending on channel mode, shall operate either in 1-in-1-out or 2-in-2-out configuration. Block placed inside Master Mixer channel shall always be in 2-in-2-out configuration, and shall perform as a mastering effects unit. Naturally there can be only one such block per voice, and because of this, it shall consume least CPU power.

All modifier locations described so far are for insert type processing, but the fourth possible modifier location shall operate in send / return configuration and in true stereo. Each Line Mixer channel shall have two send outputs and send level trims, both configurable either as pre- or post-channel fader mode. The output of aux effect modifier blocks shall be summed to outputs of master channel, and shall be eventually transmitted to the audio output ports.

In summary, there can be a maximum of 20 ($8 + 8 + 3 + 1$) modifier blocks, and $20 \times 2 = 40$ modifiers active per single voice. Real-time modulation routings shall add to the CPU load also, so overusing modifiers will certainly overrun even the most advanced PC's in market today. It should be bore in mind that the modifier architecture was designed with flexibility in mind, and most sounds do well with just a few carefully chosen algorithms at proper locations.

The modulation block shall have just one parameter which determines its *topology*. Both modifiers inside the block shall have individual common parameters, and algorithm-specific parameters and modulation routings. If a modifier is not needed, it can (and indeed should) be inactivated in order to save CPU resources. Inactive modifier shall be in bypass mode, i.e. it shall not block the signal flow. Type and subtype combination shall be used to select modifier's *algorithm*. *Balance* shall determine the amount of mix between dry (unprocessed) and wet signal applied at the output.

Filter (DCF)

Filter *mode* can be either lowpass (LP), highpass (HP), bandpass (BP), or band reject (BR). *Slope* shall define LP or HP filter order, and can take values between 6..24 dB / octave, in 6 dB increments. For BP and BR mode, slope shall be replaced by *Q* setting. *Cutoff* or *center frequency* and *resonance/Q* amount shall operate as modulation destinations.

Waveshaper

Subtype parameter shall define the preset transfer function or the function type. Other parameters shall refine transfer function's shape definition as polynomial coefficients and line segment slopes.

Delay and Reverb

A monophonic delay modifier shall have parameters for delay *time* and *feedback* level. A stereophonic version shall duplicate these for separate control on both channels, while adding *cross-channel feedback* and *panning* controls. Reverb shall be controlled via room *size*, high-frequency *damping* and stereo *width* parameters.

Chorus, Flanger and Phaser

Chorus, Flanger and Phaser modifier parameters shall consist of modulation *speed* and *depth*, delay *time* (for chorus and flanger), *feedback* level (flanger and phaser), and phaser only parameters for number of allpass filter *stages* and *sweep range*.

4.5.2 Line Mixer Section

Line Mixer section shall have eight channels C1..C8 (one for each particle), which shall be summed together into a stereo pair. Each channel shall contain controls for *level* and *pan* (or *balance* if in stereophonic mode), and two global modifier *sends* and send level and pan/balance trims. Sends can be configured to operate either in pre- or post fader mode, the latter feeding global modifier block after the output level control. As discussed earlier, each channel shall contain also an insert modifier block, which can be in a 2-channel mode if channel is in stereophonic mode.

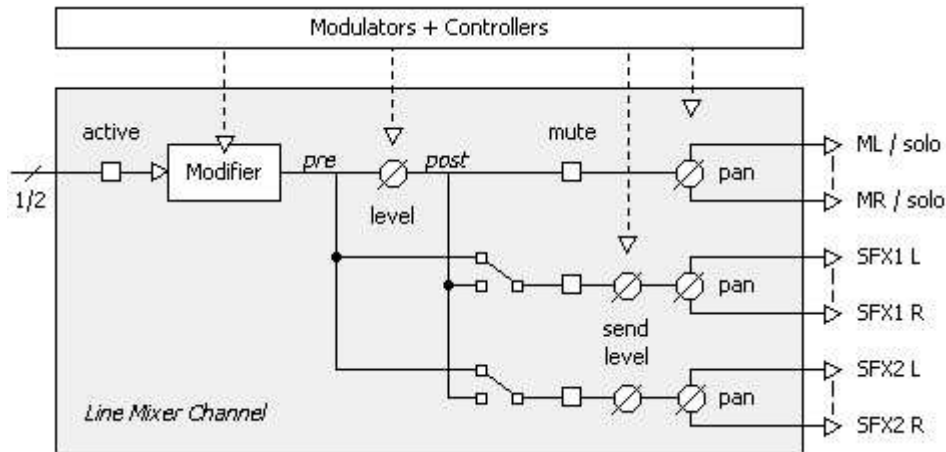


Figure 4.7. Line Mixer channel.

4.5.3 Master Mixer Section

The Master Mixer section shall have three stereo channels, which shall be summed together and routed through balance and master level trims to the audio output ports. One of the channels shall be fed with output of the Line Mixer section, and the other two contain signals that shall be fed by mixer channels' aux send streams. All three channels shall be identical, and consist of mute control, modifier block, output level fader and a balance control.

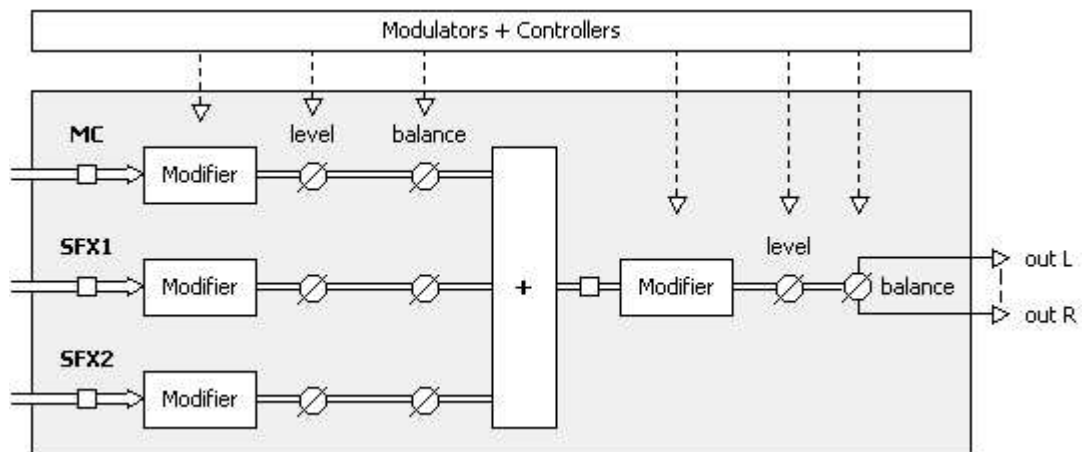


Figure 4.8. Master Mixer section.

4.5.4 Modulation Section

Modulation section shall consist of modulation generators and of a control rate modulation matrix for source to destination interconnection management. Sources shall comprise 64 EGs, 64 LFOs, 32 RVGs, all channel voice messages defined by MIDI, and 32 mouse-sensitive MMI controls. Destinations can be selected amongst 448 synthesis parameters, and the matrix shall have 40 prewired slots and 88 patch slots, giving a grand total of 128 virtual patch cords.

4.5.4.1 Sources

Envelope Generators (EGs)

EGs shall have 1..64 segments, each with parametrized *time*, *level* and *slope* settings. Time and level parameters can act as modulation destinations, and continuous slope shall parametrize whether segment is linear, exponential or logarithmic in shape. A *loop* region can be further defined between two points, or as a single point in order to realize standard ADSR style curves. There shall also be a number of predefined curves available for quick patching, and complete EG definitions shall be separately loadable from a sub-patch files.

Low Frequency Oscillators (LFOS)

LFOS shall have three characteristic parameters, which are *waveform*, *rate* and *delay*. Waveform can be selected from sine, triangle, pulse, ramp up, ramp down, random, and a short freeform WAV file that is scanned at control rate. Rate and delay parameters shall be modulatable. In addition, there shall be a parameter defining initial *phase* of waveform, *freerun* mode where LFO phase shall not be reset at trigger time, and a *fadein* time for transition smoothing. Polarity switching is done inside the modulation matrix.

Random Value Generators (RVGs)

RVG shall draw a random number when triggered. Available *trigger modes* shall be a reception of a certain MIDI event, a control rate tick, and a trailing pulse edge of an LFO. There shall also be a parametrizable *counter* that defines how many times trigger must be activated before current value is changed, or a *threshold* that controller value must exceed before triggering occurs.

MIDI Controllers

External controller events received via MIDI input port consist of <controller> and <value> pairs that can be used to control synthesis parameters in a manner similar to internal modulation generators described above. In addition to actual controller messages, other channel voice messages listed in Table 4.5 shall be available.

MMI Widgets

MMI shall contain a dedicated performance page that contains various user interface elements that can be manipulated using the mouse. These widgets can be routed to the synthesis parameter destinations in a way similar to external controllers.

4.5.4.2 Destinations

Appendix A lists modulatable synthesis parameters of PHUZOR. Destinations shall be responsible for range scaling of modulation values into parameter space, as all sources generate values that fall into [0..1] range. They shall also be able to interpolate values between control rate updates, in order to reduce zipper noise and sharp transient clicks

that would otherwise be inherent in abrupt changes. For easier patching, there shall be conceptual destination groups, where one virtual destination is connected to multiple actual destinations (e.g. particle pitch destination group affects pitches of each individual particle from P1 to P8).

4.5.4.3 Modulation Matrix

All modulation sources shall be routed to their destinations via modulation matrix, consisting of 128 slots or virtual patch cords for those connections. Each cord shall have individual settings for *source*, *destination*, *amount* and *curve*. Amount shall be modulatable, and can be negative in order to change source polarity. Curve selection shall be destination dependent, and there shall be linear, exponential, logarithmic and various quantized variations available.

There shall actually be two control rate modulation matrices in PHUZOR. The first one shall contain essentially hardwired routings that are frequently used, but seldom changed. Routing ‘note on’ message’s key value to particle’s pitch, or amplitude EG connection to DCA level are examples of prewired matrix cords. These settings can be overridden in the second matrix, which contains all patch specific modulation routings. Both matrices shall be independently loadable from customary files. Synthesizer template files shall also contain a specific modulation matrix section so that the routings available in the original model can automatically be included.

4.6 Specific Requirements

4.6.1 Communication Interface Requirements

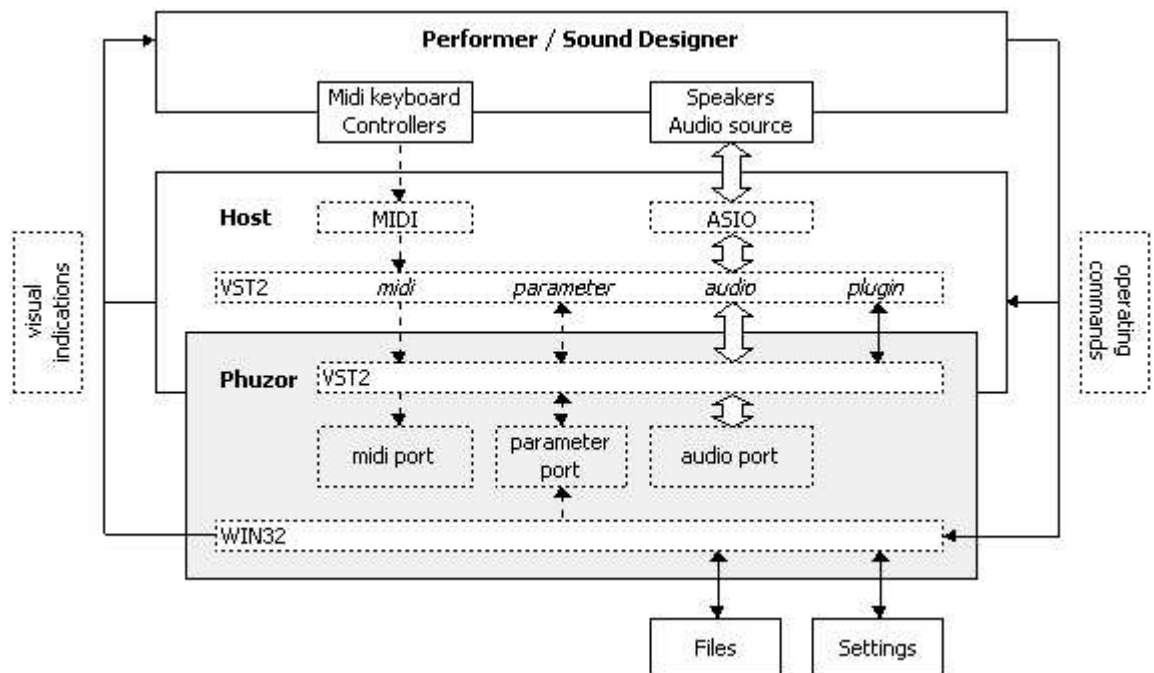


Figure 4.9. System Context Diagram with interfaces and data stores.

System context diagram of Figure 4.1 is given in more detail in Figure 4.9, where physical and logical interfaces and data stores have been added. Communication interfaces shall consist of MIDI and ASIO, which are described in this section. Software interfaces shall include VST2 and WIN32 APIs, which are covered in subsequent sections.

4.6.1.1 MIDI

Real-time controller signals are translated into MIDI protocol [26] by external MIDI devices, and transmitted to PHUZOR's MIDI input port via VST2 interface. These signals shall include gates to start and stop audible sound at desired pitch, and various articulation messages in form of switches and continuous controller values.

Table 4.5. MIDI implementation chart of PHUZOR, version 1.0, 08 / 2005. 'Transmitted' column has been left out, as there are no MIDI messages generated by this device.

Function		Recognized	Remarks	Global Parameter	Range
Basic Channel	Default Changed	1..16 1..16	memorized	01 Midi Channel	1..16
Mode	Default Messages Altered	Mode 1,3 Mono, Poly, Omni on/off Mono -> Omni	memorized	02 Midi Omni	on / off
Note Number		o : 0..127	⌘	03 Tuning Table	nip
Velocity	Note on Note off	o : v = 1..127 o	⌘ □ ⌘ □	04 Velocity Curve 04 Velocity Curve	nip nip
After Touch	Key's Channel's	o o	⌘ □ ⌘ □	05 Aftertouch Curve 05 Aftertouch Curve	nip nip
Pitch Bender		o (14 bit resolution)	⌘ □	06 Bend Ranges	nip
Control Change	0..63 64..95 96..101 102..121	o (14 bits) o (7 bits and switches) x (rpn + nrpn) o (7 bits)	⌘ □ ⌘ □ (>= 64 : on) nip ⌘ □		
Defaults	1 7 10 64	o : mod wheel o : main volume o : pan o : sustain pedal	-> voice LFO amount -> master level -> master pan -> EG sustain		
Program Change		x			
System Messages		x			
Aux Messages	Local control All notes off Active sense Reset	x o 123..127 x x	□		
Notes	o : yes x : no ⌘ : can be routed via modulation matrix □ : routed via global midi input filter, memorized nip : not in prototype version			Mode 1 Omni on, Poly Mode 2 Omni on, Mono Mode 3 Omni off, Poly Mode 4 Omni off, Mono	

4.6.1.2 ASIO

ASIO is an acronym for Audio Stream Input Output, and it defines a software and hardware layer for real-time exchange of multi-channel digital audio data. It replaces operating system's low level audio handling methods with functions that are tied even closer to the hardware, thus providing lower latency audio data streams. PHUZOR shall not be directly concerned by this interface, as it uses VST2 also for audio streaming, and

it is included in the diagram for completeness. More on the subject can be found in [37]. ASIO drivers are strongly recommended instead of Windows MME drivers for practical real-time use.

4.6.2 Software Interface Requirements

PHUZOR shall run within Microsoft Windows OS 98SE or XP Home / Professional editions, and it shall be a DLL (dynamic link library) extending a VST2 compatible host application. In order to fit into the host's VST2 slot, corresponding plugin interface shall be implemented in the DLL.

Although PHUZOR shall be developed with multi-platform portability in mind, certain operating system services have to be utilized. Where possible, higher level libraries like ANSI stdio or C++ streams shall be used instead of hardwired OS functions. Microsoft's WIN32 API [45] is a collection of interfaces and libraries that allow OS -related calls to be made, more specifically those dealing with file IO, registration database and graphical user interfaces shall be routed via standard WIN32 API.

4.6.2.1 VST2 Host

VST2 plugin standard was discussed earlier in section 3.3.3. For the purpose of PHUZOR, a subset of methods in the SDK shall be utilized, and can be categorized as *a)* core functionality providing basic process controlling actions (loading, initialization, termination, state changes and MMI functionality), *b)* MIDI event routing methods, *c)* parameter automation interface and *d)* audio stream routing. Definition for all VST2 methods is documented in [37].

4.6.2.2 Files

Files shall be used to store oscillator waveform data as samples (RIFF WAV and SF2 formats), or as spectral description (customary SPE format) and synthesis parameter setups (patches or subpatches). Files can be accessed via MMI at any execution stage between initialization and termination. Initial state of synthesis parameters shall be loaded during initialization time from specially named default patch file, which shall be structurally like any other patch file, allowing customizable startup settings for patch editing.

RIFFs as WAV Files

Microsoft's RIFF (Resource Interchange File Format) specification defines a chunk-based file format capable of storing different multimedia content, including data for digital audio and video. The container files house also Windows' native sound WAVs, supporting various bit resolutions, sample rates and encoding formats. Multichannel audio is represented as interleaved streams. RIFF documentation can be found in [45] and a fine WAV-oriented presentation is at [46].

PHUZOR shall be able to load WAVs with any sample rate, but is limited to bit resolutions 8 and 16, and to linear uncompressed PCM sound data format only. One or two channel data shall be supported. These fundamental parameters describing WAV structure are stored in 'fmt ' chunk, which is usually located at the very beginning of the file. Actual sample frames are inside 'data' chunk following the format definition, with left-justified sample points stored in little-endian order. Looping information is read from 'smpl' chunk, but only one loop segment shall be supported¹. All loop types currently defined in specification (i.e. forward, alternating and backward only) shall be recognized, however. Fine tuning information is stored inside 'inst' chunk. Other chunks shall be ignored, should there be any.

RIFFs as SoundFont 2 Files

The problem with WAV files is that they can conveniently contain only a single sample. E-Mu addressed this shortcoming with a RIFF -based format called SoundFont [23], which bundles a collection of individual samples into one compound file, while adding envelopes and LFOs to enhance otherwise passive waveform playback with basic modulation related parameters. The waveform collection can be assigned to different key and velocity zones so that different samples can be triggered instead of pitch shifting a single waveform to entire keyboard range.

SoundFont files consist of three basic chunks, each further divided into variable length list of subchunks. 'INFO' chunk defines supplemental information for documentation purposes, and is not used in actual sound production stage. 'sdta' holds all binary sample data as a single large pool of samples, and 'pdta' corresponding preset, instrument and sample header related information of that data. PHUZOR shall utilize sample data and sample header chunks, together with loop and zone related parameters. Filter, modulation source and effect parameters could also be loaded, but that work has been shifted towards later releases.

Spectral Definition Files

Spectral composition of a single-cycle oscillator waveform can be defined using a proprietary file that is stored in human-readable format. It contains a snapshot of sound's steady-state spectrum, possibly using multiple fundamental frequencies in order to spread the waveform realistically across the whole keyboard range. By convention, files shall have 'SPE' extension and can contain only printable ASCII characters.

An SPE file shall consist of sections delimited by lines having square bracketed [section name]. Inside each section there shall be attribute lines having a keyword, an equal sign and the attribute's value. All characters followed by an apostrophe (') shall be treated as

¹ This is the loop segment where playback repeats itself when envelope is in its sustaining stage. Some hardware samplers have also separate release loop segment, but this is not a very commonplace feature.

comments and shall be ignored. Empty lines and white space shall be allowed between tokens to add clarity.

There shall be two types of sections. [Header] shall define name of the waveform, the maximum number of groups that partials are accumulated into, and the number of pitch sections to follow. Each analyzed pitch shall have its own section, and shall be identified by a note name from equal-tempered scale, and its octave. For example [A5] defines pitch of A above middle-C, usually tuned to 440 Hz. Inside pitch section there shall be one line for each partial with a keyword for partial number (0 describes attributes for fundamental frequency). After equal sign, there shall be values for group (starting from 1), frequency, amplitude and phase.

PHUZOR shall support up to 8 groups, and up to 128 pitches. Prototype version shall not interpolate between spectra of different pitches, so spectral definition for one pitch shall be valid for all interleaving pitches until new pitch section is encountered. It should also be noted that although pitch section maps spectra to 12-tone equal temperament keys, any tuning scale can be defined by setting fundamental frequencies to desired values.

Synthesis Parameter Files

Traditionally, all synthesis parameters constituting a synthesizer timbre are collected into a single patch, and a group of these equal-sized patches are collected into a bank fitting exactly into the synthesizer's onboard user program memory. Due to versatile synthesis architecture, PHUZOR can be controlled using a large array of parameters, and because redundant information from patch to patch is often included, slightly different approach shall be utilized.

Each PHUZOR patch shall consist of a number of subpatches. If a parameter value is not defined in a patch file, its definition shall be read from a subpatch, and if it still not found, a hardcoded default value shall be used. On the other hand, any parameter value defined in patch scope shall override corresponding value defined in subpatch scope. On positive side, patch programming process shall become faster and more intuitive, as previously developed subpatch blocks can be assembled together and there is no need to program each parameter separately. A fine tune in common subpatch block shall have influence on all patches sharing that block. Patch sizes shall become also smaller. Negative side effects arise from similar reasons. A tweak in shared block might cause an unwanted change in a specific patch, and it is also more difficult to transfer patches between different locations as multiple files need to be included. A solution to these problems is to include an option to save all parameters into single patch (thus overriding all possible subpatch definitions).

All files storing synthesis parameters shall have a common format, where each file shall be divided into number of sections, and each section shall be divided into number of attributes and their values. Syntax shall be similar to the one described for spectral files above.

Table 4.6. Subpatch files.

Subpatch	Category	Ext	Description
Template	Global	syn	Prewired ARM and CRM routings (e.g. Roland Juno-2)
ARM Matrix	Modulation	arm	ARM routings (e.g. DX7 algorithm)
CRM Matrix	Modulation	crm	CRM routings
CRM Macro	Modulation	mac	Modulation destination group (e.g. brightness of all particles)
Envelope	Modulation	env	Envelope curve
LFO	Modulation	lfo	LFO settings
RVG	Modulation	rvg	RVG settings
Particle	Source	par	Entire particle or particle group (e.g. FM operator stack)
DCO	Source	dco	DCO block
DCA	Source	dca	DCA block
DCW	Source	dcw	DCW block
Modifier	Modifier	mod	Single modifier or pair of them
Waveshaper function	Modifier	shp	Waveshaper transfer function
Line Mixer channel	Mixer	mix	single or multiple line mixer channels
Master mixer channel	Mixer	mas	single or multiple master mixer channels (including modifiers)

4.6.2.3 Registry

PHUZOR shall read user-definable preference information from Windows registration database and adapt itself to those settings during initialization time. A dedicated preference dialog shall be included in MMI to allow easy preference configuration.

4.6.3 Man Machine Interface (MMI)

PHUZOR shall have a single main window with switchable pages to navigate through different parts of the interface. Only modal dialogs, such as alerts and confirmation prompts, and standard OS file dialogs shall appear outside of the main window. The main window shall be divided into 4 parts, as shown in figure 4.15 below.

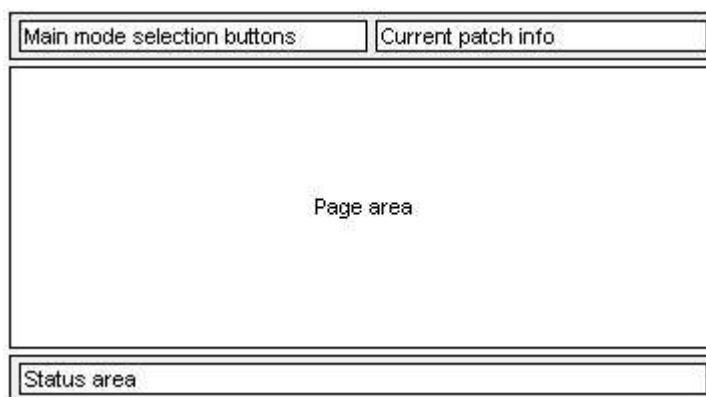


Figure 4.15 Main window components

The top part of the window shall be always visible, and shall contain buttons for main mode selection and currently active patch area. Below that there shall be a space for individual page and its controls, and whenever switching to a new page, contents of middle part shall be changed. Bottom part shall contain a status row, which shall be always visible like the top row.

Table 4.7. Pages and panels.

Page	Description
Perform	Performance mode
Widgets	MMI controls like xy-pad, sliders, knobs, buttons attached to parameters or macros
Library	Patch database handling, not included in prototype
Design	Sound Design mode
Structure	ARM Matrix with audio signal mixing controls + modulation panel
Source	Panels for ARM, DCO, MFX, DCW, MOD and DCA
Mixer	Interface to Line Mixer + MFX and MOD panels
Master	Interface to Master Mixer + MFX and MOD panels
Modulation	CRM Matrix
Configure	Settings MMI

4.6.4 Performance Requirements

Controller data shall be transmitted via MIDI. Host may transmit sequencer generated data at a higher tick resolution than is defined in MIDI standard, but adhering to serial line's transmission times, event shall be available in timestamped buffer of PHUZOR after 1 ms of its initial generation.

Changes in patch parameter values shall be reflected in synthesized audio. Their effect and treatment shall be similar to those of sound engine's internal modulation generators, which shall be evaluated at control rate, with linear interpolation for in-between values. Control rate of PHUZOR shall be the same as block rate of audio, and setting block size to 512 samples at 44.1 kHz sampling rate gives 11.6 ms per block. So in general, PHUZOR shall update parameter value in 12 ms or less. Parameter nature affects the speed of the update, for example if file IO is involved when changing oscillator waveform, the response time is dependent on size of file and the speed of storage media.

Synthesizing continuous audio stream means that sound engine must be able to fill output sample buffer at the rate that those buffers must be fed into audio hardware. Using figures from previous paragraph, sample period is 0.0226757 ms, so PHUZOR shall be able to calculate single sample value in 22.68 μ s or less. This is the absolute maximum value though, as host, other plugins, and even other parts of PHUZOR do need their slice of CPU time. Audio input processing takes place at the same rate and is included in value above.

Empirical tests have shown that when latency times exceed 20 ms, the playability of the instrument begins to suffer and it starts to feel unresponsive, and latency times over 50 ms will make it inappropriate for live performance. This is a subjective matter however, and is influenced by performing style, physical properties of external controller device, the type of the sound and so on. If requirements stated above can be realized, the 20 ms limit is achievable.

4.6.5 Hardware Interface Requirements

PHUZOR shall work with standard PC having Intel Pentium 4 processor clocked to 2 GHz as a minimum. Other SSE2 instruction set processors, like Intel Xeon and AMD Athlon 64 can be used alternatively.

Minimum of 256 MB of RAM is required, but 512 MB is recommended. Full installation of PHUZOR will take 1 MB of hard disk space. Patches and particularly sample data add to this required size. MMI requires 16-bit color depth, and screen resolutions starting from 1024 x 768 pixels.

PC should be equipped with Windows MME-compatible audio hardware, but for performance reasons, ASIO compatibility is recommended. At minimum, standard 16-bit / 44.1 kHz sound card or mainboard integrated AC'97 compatible audio chipset is sufficient. The prototype shall not support sample data beyond 16 bits, but as internal processing resolution is 32 bits, and as sample rate can be 96 kHz or even higher, sound quality can be improved using 24-bit / 96 kHz audio hardware. For external controller setup, MIDI interface is also required.

4.6.6 Testing Requirements

Following items shall be analyzed and discussed when evaluating results of this thesis (chapter 7) :

- Sawtooth waveform (stock, sample, spectral, supersaw)
- FM electric piano patch from Yamaha DX7 as emulated by PHUZOR, with a discussion of the theoretical spectrum.

Responsiveness to external MIDI device shall be tested. Both subjective and quantitative measurements of timing performance are carried out. Flexibility vs. complexity of synthesis architecture shall be discussed, and observations on hierarchical patch management system shall be made.

4.6.7 Portability Requirements

Platform specific code shall be separated from portable code by interfacing classes, header files and cross platform libraries. Plugin specific portions shall also be made easily replaceable using abstract framework classes. Specific processor dependent code optimization techniques shall be bracketed by compiler directives, and a portable non-optimized alternative shall always be available. Future porting options include DirectX and AudioUnits architectures, and Macintosh X operating system.

4.6.8 Safety Requirements

PHUZOR can cause permanent damage to hearing and speaker systems, even if it is working properly. Sound volume levels should be kept within reasonable limits at all times, as simple change in one of the synthesis parameters can have unthinkable side effects within other parts of programmed patch. Also, in case of software malfunction, loud random sounds may be outputted, and the only way to silence them is to turn the volume down and reboot the computer.

A panic button shall be accessible from MMI, and pressing it shall gracefully mute all voices that are currently audible. Gracefully here means that a hardcoded emergency envelope tail (i.e. one that fades out very quickly, but does not cut sound off immediately in order to avoid sharp transient clicks) shall be run through both outputs. To reduce sudden amplitude changes in proper working situations, envelopes shall not have brickwall steps, but even at minimum settings do employ a short constant transition time. EGs shall be fine-tuned so that they still feel responsive despite the short minimum time interval.

PHUZOR shall also respond to ‘all notes off’ commands received via MIDI. The action is to perform emergency muting as described above, and to reset voice allocation algorithm to power on state.

Chapter

5 Architectural Design

This chapter describes the software architecture design phase of PHUZOR so that all requirements given in chapter 4 are implemented. It takes a topdown approach by decomposing PHUZOR into packages, subsystems, classes, attributes, methods and their interfaces. This enforces information hiding and enables independent design of subcomponents without affecting others.

A package is a compile-time concept, while subsystems are considered as run-time entities. Subsystems are assembled into a collection of classes that are interconnected by associations and other static relationships. Classes are further decomposed into methods and attributes, which is the lowest level of the architectural design phase. Data flow and messaging between subsystems and classes is conducted through a collection of well-defined interfaces.

Section 5.1 gives a more detailed view of PHUZOR in its relation to external subsystems. Section 5.2 describes top level subsystems and packages in framework context, while 5.3 discusses some common design issues. Finally 5.4 is ready to give a detailed view of the subsystems' internal composition.

5.1 System Context

Figure 5.1 extends the system context diagram of Figure 4.9 further by describing the data that is passed between external subsystems, data stores and PHUZOR. Performer - Host connection is omitted for clarity. By examining the figure below, some architectural design decisions become evident.

To reduce CPU load and complexity of implementation, erroneous or out-of-context input data should be discarded as early as possible. This can be achieved by introducing filtering and state awareness inside low-level input components. For example, rudimentary MIDI filtering shall ignore all events addressed to alien channels, and audio processing can be reduced to mere output buffer clearing if there are no pending events.

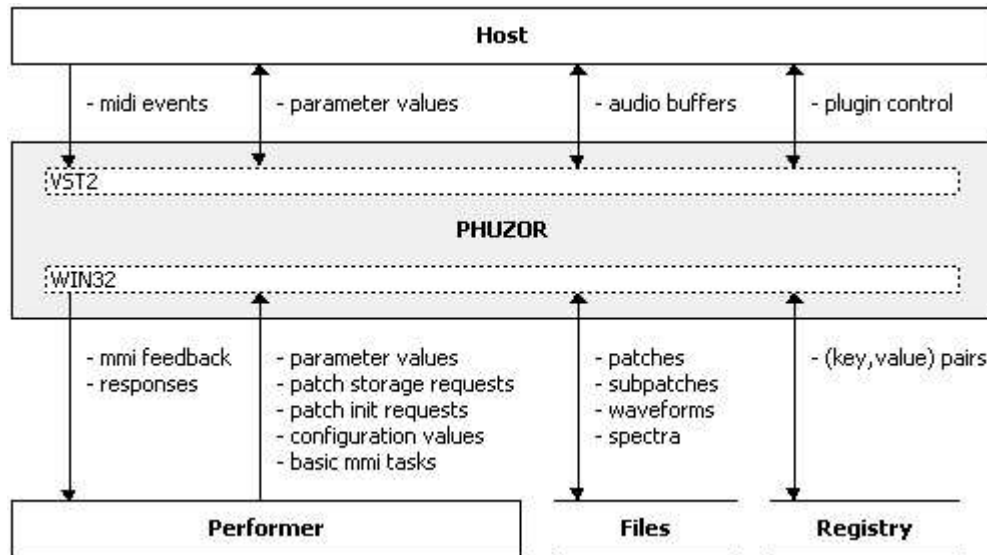


Figure 5.1 System Context Diagram in DFD format.

Synthesis parameters are updated either by MIDI events, host's parameter automation or by performer's MMI actions. This suggests that a single component is responsible for the actual update, and that data from each source should be parsed and converted to a format that the updater component understands. Configuration data comes directly from the registry or as a result of MMI editing actions, so MMI component should have the capability of converting that data into editable form and submitting editing changes back to the registry interface implementer component. Furthermore, the MMI component takes part in all tasks between PHUZOR and performer.

File handling should be centralized to a single service component, which shall be accessible from all other subsystems needing its services. This would include low-level IO, syntax checking and conversion actions leaving only semantical considerations to the invoking instance.

Plugin process control and host service interface should also be centralized and accessible from any part of PHUZOR. Finally, the actual synthesis engine should be separated from the rest of the code in order to make it replaceable by another engine, and conversely to make port for different plugin architectures possible by just changing relevant parts in the framework.

5.2 System Design

At the topmost level, PHUZOR is a dynamic link library (DLL), which is a single module that can be loaded into the memory on demand. This chapter decomposes PHUZOR into packages and top level subsystems, and keeps the scope on framework (i.e. reusable) layer, and leaves the discussion of proprietary synthesis architecture level details to section 5.3.3.

5.2.1 Subsystem Decomposition

PHUZOR is decomposed into 13 top-level subsystems shown in Figure 5.2. Dashed boxes denote packages. Subsystems belong to both framework and extended packages, as framework classes are often overridden to provide custom functionality.

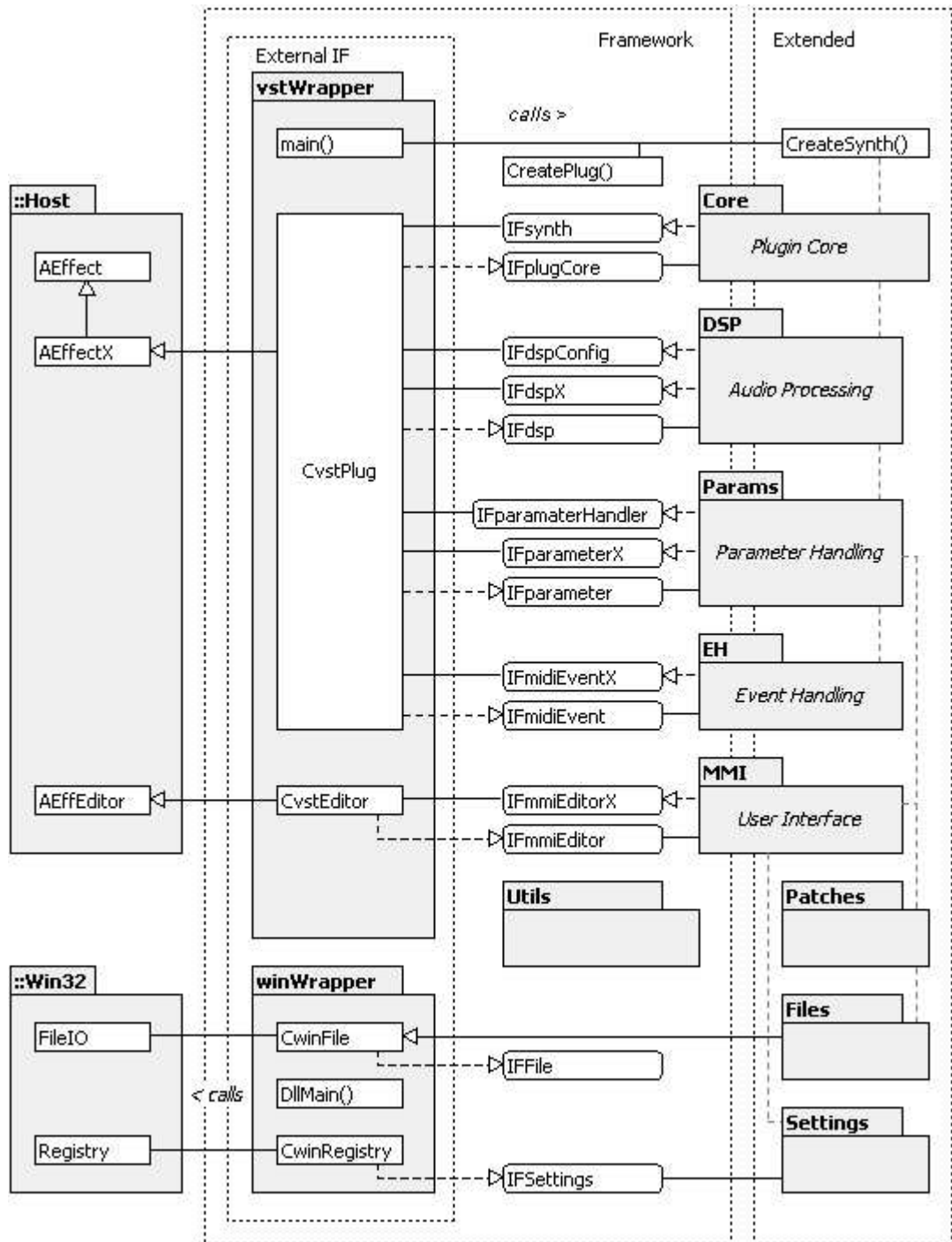


Figure 5.2 Subsystem layer of PHUZOR.

5.2.2 Subsystem Responsibilities and Interfaces

Table 5.1 lists PHUZOR subsystems. Left column contains subsystem names, components and main interface classes, and right column gives a short description of their responsibilities. Interfaces between Extended subsystems are described later.

Table 5.1 Subsystem responsibilities and interfaces.

::Host	Encapsulates VST2.3 SDK
AEffect	VST1 plugin functionality (abstract class)
AEffectX	VST2 plugin functionality (abstract class)
AEffEditor	VST MMI core functionality
::Win32	Encapsulates relevant parts of Win32 Platform SDK
FileIO	Basic file handling functionality
Registry	Registration database functionality
vstWrapper	Hides VST from rest of the implementation
CvstPlug	Wraps native VST calls for framework
CvstEditor	Wraps native VST core MMI calls for framework
winWrapper	Hides Windows specific implementation
CwinFile	Base class for file handling
CwinRegistry	Base class for configuration database
Core	Plugin core functionality (like creation and mode changes)
IFsynth	mode changes, plugin properties
IFplugCore	audio port creation, plugin properties
DSP	Audio processing
IFdspConfig	audio related properties (sample and control rate, blocksize, tuning)
IFdspX	audio processing
IFdsp	global audio stream enabling and disabling
Params	Parameter handling
IFparameterHandler	audio related properties (sample and control rate, blocksize, tuning)
IFparameterX	audio processing
IFparameter	global audio stream enabling and disabling
EH	MIDI event handling
IFmidiEventX	event processing
IFmidiEvent	global event stream enabling and disabling
MMI	Man machine interface
IFmmiEditorX	opening, closing, updating of MMI windows and controls
IFmmiEditor	update and resize requests
Utils	Static utility functions and macros
Patches	Parameter chunks
Files	Loading and storing of Patches and waveform definitions
Settings	Configuration management

5.2.3 Interaction Diagrams

Figure 5.3 shows the Controller Thread management of Figure 4.4 in subsystem and class levels, in relation to the real-time DSP synthesis thread. The top part graphs the Core actions that need to be taken so that the Host is able to stream MIDI events to the

plugin. Letters A and B refer to section 6.2, which describes the algorithm in more detail.

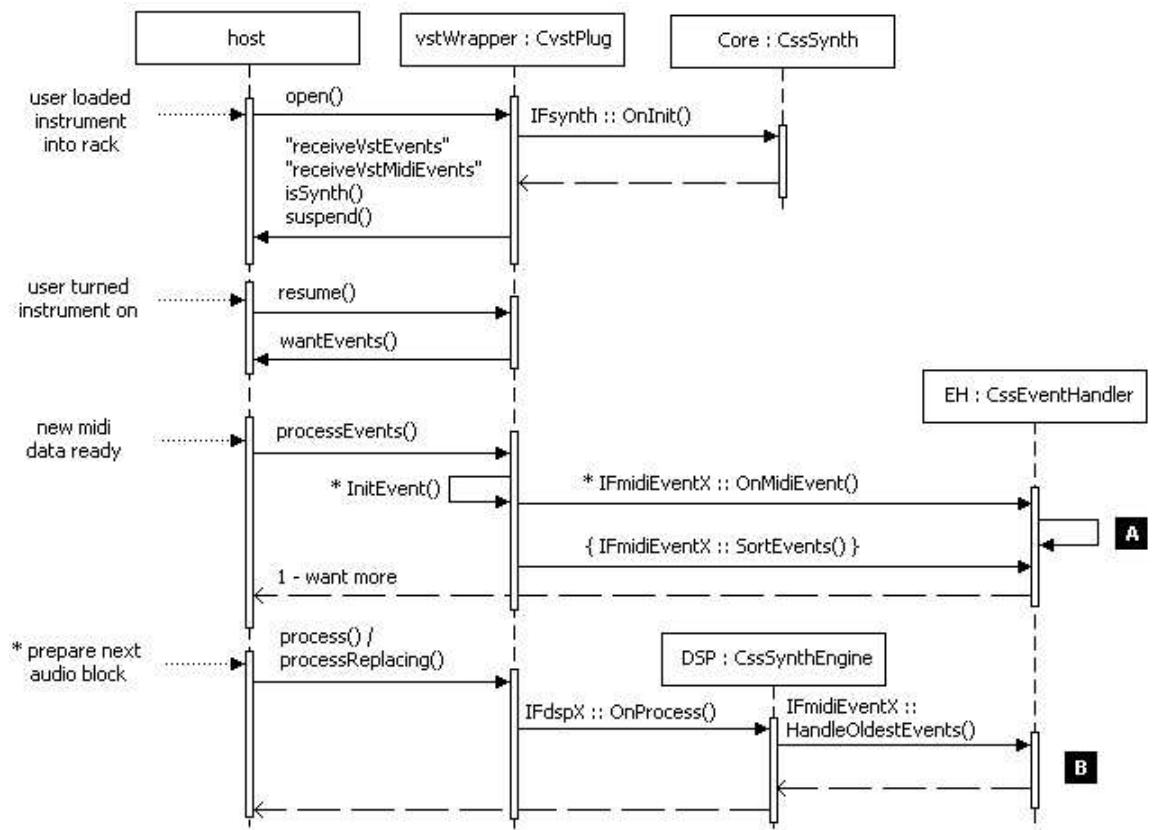


Figure 5.3 Reception of MIDI events and their processing.

5.2.4 Subsystem Division, Referral and Concurrency

Each class belongs to one of the three packages, namely External Interfaces, Framework and Extended. This approach fulfills portability requirements by allowing classes in External Interfaces package (CvstPlug, CvstEditor, CwinFile and CwinRegistry) to be replaced with ones for another plugin architecture or operating system. Separating Framework and actual project specific code into different packages allows reusing much of the trivial core functionality and rapid implementation of totally different synthesizer architectures.

On subsystem level, division is based on functional decomposition and concurrent threading model. As described in section 3.3.4, audio, events and remaining functions are run concurrently in different threads, so DSP and EH subsystems are a natural consequence of this model. Functional composition is further used to divide rest of the functionality into Core, Params, Patch, MMI, File, Utils and Configuration Management subsystems.

Referral between subsystems can only be performed via interface classes defining the subsystem interfaces, each of which provides a global way of accessing its implementer (subsystem container has also static convenience methods that handle the casting

internally). This is possible as only one subsystem instance can exist concurrently. In time critical situations it is permissible to store pointer to the interface implementer in order to reduce the method call overhead. This poses a limitation that interface implementers can not change at runtime, but in context of PHUZOR this is not applicable anyway.

5.3 Common Design Issues

Global hardware resources are managed by the Host application, so all synchronization, thread prioritization, and global buffer management actions are handled therein. The software control implementation shall be event-driven, dispatched by Host and OS, implying that the plugin MMI must be modeless.

Plugin shall run only within Host context, so stand-alone version shall not be developed. The architectural construction shall reflect real-time responsiveness to external interface generated inputs. Because of this, PHUZOR shall be optimized for speed over memory, and a slight overhead in the initialization phase (caused by stock waveform creation and MMI component setup times) shall be acceptable for improved real-time performance. MMI shall only be available in English language.

5.3.1 Handling of Boundary Conditions

5.3.1.1 Initialization

The main entry point into PHUZOR is the `main()` function, which is located inside `vstWrapper` subsystem, and called by the Host when plugin is loaded into the instrument rack. It shall create and initialize an instance of the `CvstPlug` class (to interface the Host application), and thereafter the actual `CssSynth` derivative (which acts as the framework main class and subsystem container). All other subsystems are created and initialized by `CssSynth`. This includes the MMI components (as discussed above), so that they are ready to be displayed when the Performer opens the instruments front panel. In the class level, each class shall have an `Init()` method that should be used for operations such as memory allocation, and other failure sensitive initialization tasks.

5.3.1.2 Termination

Host starts the termination sequence by invoking a `close()` call, thereafter deleting the `CvstEditor` and `CvstPlug` instances. `CvstPlug` in turn deletes the framework's main class `CssSynth`, or its descendant created in initialization phase. The main class is the owner of all other subsystems, and responsible of releasing the memory allocated for them.

5.3.1.3 Failure

Special consideration shall be given to failure handling, as malfunction in form of loud signals might cause permanent damage to listeners' ears and audio equipment speakers.

Software errors can be minimized by systematical null pointer testing and exception raising, but due to the complexity of the algorithms, it is very difficult to achieve a high rate safety level. For these reasons, an audio fuse (which mutes audio output when a certain predefined number of out-of-range samples have been generated, and restricts the output inside system's dynamic range) shall be patched before main outputs. There shall also be a panic button that is accessible any time, allowing rapid emergency muting of output.

5.3.1.4 Mode-specific Behaviour and Mode Changes

PHUZOR shall be either in suspended or resumed state. In suspension, all audio and MIDI input is ignored, and the only action that is performed is the zeroing of the output buffer. In resumed state, all input streams are active, and synthesized audio is written into the output buffer.

PHUZOR shall initially be in suspended state. Switching between resumed/suspended state is initiated by Performer's MMI actions, which the Host typically metaphors to power on/off toggling. Suspended state is also entered if the audio fuse blows up, or if the panic button was clicked by the Performer.

5.3.2 Implementation and Testing Environment

The implementation and testing phases are conducted using following hardware and software environment:

Table 5.2 Implementation and testing environment.

Hardware	
Processor	Intel Pentium 4, 2.4 GHz, FSB800, 512 KB L2 cache
Memory	512 MB, DDR400
Hard Disk	120 GB, DMA/133, 7200 rpm
Audio Card	SB0220 Live, EMU10K1-JFF
Audio Equipment	Basic home stereos + headphones
MIDI Controllers	5 octave keyboard + pitch bend + modulation wheel + sustain pedal
Software	
OS	Windows 98 SE
VST Host	MiniHost 1.04 [50]
Compiler	Microsoft Visual C++ v6 with MFC 4.2 and Win32 SDK [45]
Audio Editor	Sonic Foundry Sound Forge v4.5 [51]
Spectrum Analyzer	Sonic Foundry Sound Forge v4.5, MATLAB [59]
Sound Driver	ASIO4ALL v1.8 [52]
Plugin SDK	VST 2.3 SDK [37]

5.3.3 Extended Design

PHUZOR shall be implemented through framework's Extended package class inheritance, and there shall thus be no direct connection to the Host SDK. External Interfaces package is also implemented in framework, but in rare situations their direct use is possible as was described in 5.2.4.

5.4 Detailed Subsystem Descriptions

This section describes the inner construction of each subsystem, based on functional grouping. Using this approach makes it possible to visualize components of framework and PHUZOR in a single diagram, and to describe relationships between classes more clearly as there are less loose ends than there would be if each subsystem were drawn into a separate diagram. Downside is that some classes are split up into more than one diagram (particularly `CvstPlug`).

Whenever a class outside functional scope is included in the diagram, its name is preceded by scope delimiters (`::`) and drawn over light gray background. Bold class name is used when framework classes are inheritable, and dashed outlines denote the interface classes. Note that only the most important methods and members are shown, more detailed design is in [53].

5.4.1 Core

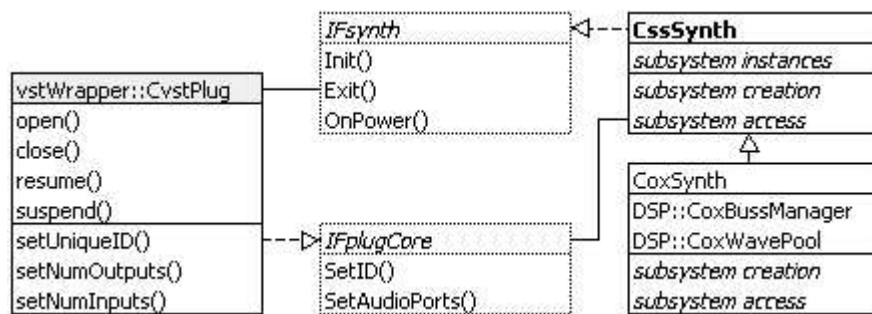


Figure 5.4 Core classes. Host dispatched message handlers are shown on left.

Core classes manage initiation, termination and mode change functionality, and act as containers of other subsystems in framework and PHUZOR. The main entry point creates `CvstPlug` and `CoxSynth` instances as discussed in 5.3.1.1. The audio port topology is fixed during initialization phase (VST does not support dynamic port configurations). Two DSP related global subsystems are also created inside Core, and are responsible for fixed audio buss structure management, and dynamic oscillator waveform storage. `OnPower()` is invoked in response to suspend/resume calls, and shall be routed to EH and DSP subsystems for input port (de)activation tasks.

5.4.2 Event Handling (EH)

MIDI input port is enabled and disabled using `EnableEventStream()` method, which is called from `OnPower()` message handler. `processEvents()` is called when there are new events waiting to be processed, and it routes them to `CsxEventHandler` class for input filtering and buffering (`m_cEvents` is a time-stamped MIDI event queue).

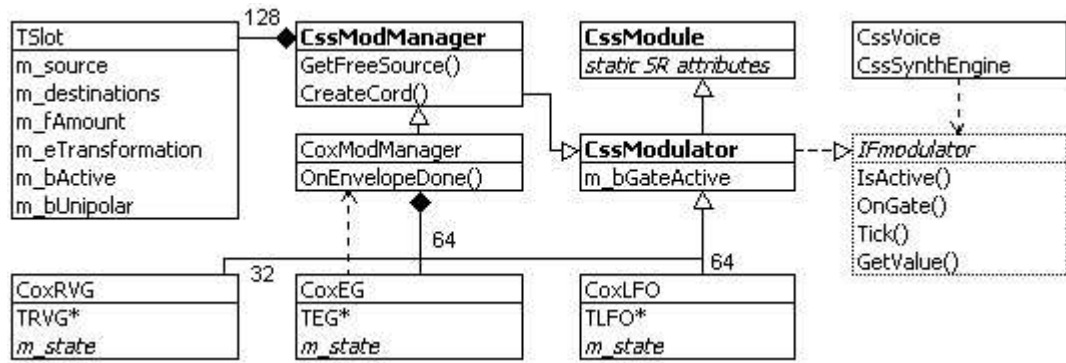


Figure 5.8 Control rate modulation classes.

Control rate modulation matrix is contained in `CsxModManager`, and implemented as an array of `TSlot` instances. Gate and tick signals originate from `CsxSynthEngine` and `CsxVoice` classes via `IFmodulator` interface, which is implemented in `CsxModulator` descendants. `CoxEG` notifies the system after finishing its release segments.

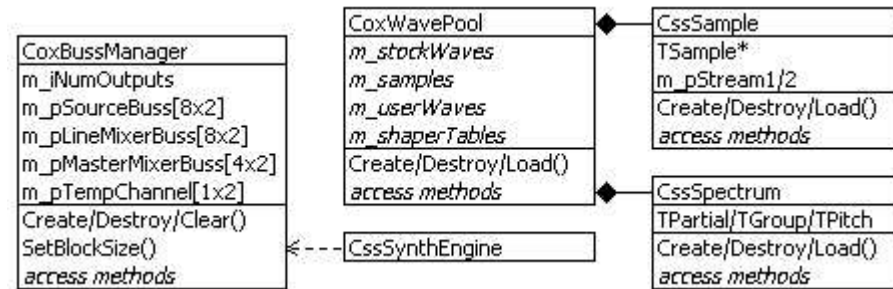


Figure 5.9. Static DSP classes.

Finally, there are two statically allocated container classes managing *a)* the busses that connect main synthesis components together, and *b)* the precomputed and sampled waveforms loaded from disk storage. All busses are stereophonic and are created in response to host supplied maximum block size notification message.

5.4.4 Parameters and Patch

The final release version of the framework shall have a uniform parameter handling mechanism, which simplifies synthesizer development by reducing the parameter definition phase to mere listing task. Further benefits are gained because provided list is enough to facilitate parameter automation, interpolation, modulation, patch file and MMI related interaction management.

However, the prototyped version of PHUZOR uses a tailor-made parametrization scheme without host automation features. Parameters are stored inside `CoxPatch`, which defines entire synthesizer timbre, and there is only one instance available at one time (PHUZOR is monotimbral). It is contained inside `CoxParameterHandler`, which has also a map for routing controller messages when programming a patch, and two methods for storing

MMI consists of three kinds of basic components. `CwndPage` instances are used to fill the center part of main window acting as containers for other pages and `CwndPanels`, which are smaller areas inside a page, grouping related section components together (e.g. DCO or Modifier controls). Interaction is handled using a set of custom widgets, which are usually assigned to single parameter, or to a group of them (e.g. graphical EG editor). Widgets restrict parameter editing to permissible ranges, and each editing action is reflected to the model via centralized `IFParameter::OnParameterChanged()` interface. Parameters are transferred into widgets within `ShowParameters()` calls.

`CccSkin` class is used by all visual MMI components, and it provides customizable color schemes for the interface. Predefined schemes are stored in `Settings`.

5.4.6 Files

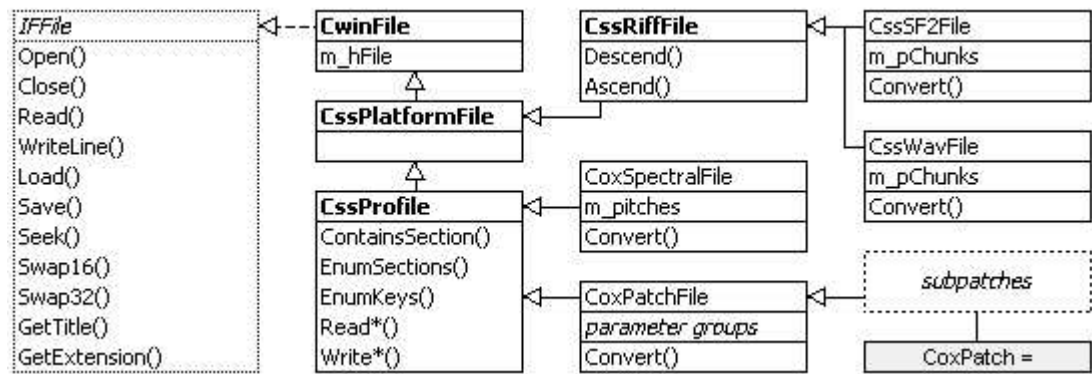


Figure 5.12 File classes.

File classes are created on demand, i.e. a temporary instance of proper type is used to transfer data between disk and memory. `CcssPlatformFile` is platform independent class that for the purpose of PHUZOR is derived from `CwinFile`, which implements basic file IO methods published in `IFFile`. `CcssRiffFile` and `CcssProfile` implement the functionality that is common to the sample data and text-based proprietary spectral definition and patch files. Instantiable classes are `CcssWavFile` and `CcssSF2File` for sample data, and `CoxSpectralFile` and subpatch file types listed in Table 4.6. Each of these defines the `Convert()` method for translating stored data into/from memory structures.

`CoxPatch` has an assignment operator for each converted subpatch type, and MMI is used to initiate loading / storing of all instantiable file classes (links omitted from picture above for clarity).

5.4.7 Settings and Utilities

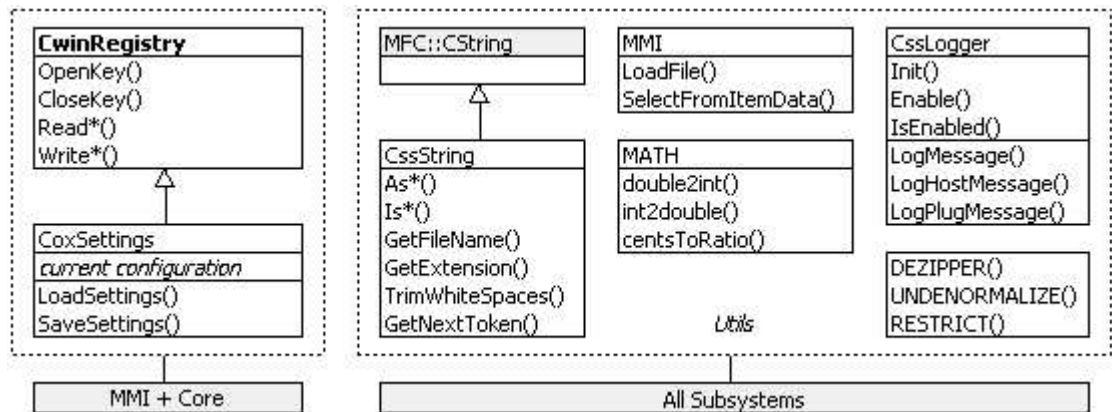


Figure 5.13 Configuration management and utility functions, macros and classes.

`CoxSettings` is a static class that uses `CwinRegistry` to read and write <name,value> pairs from system registration database. Utilities subsystem consists of `CssLogger` which is used to log messaging between host and plugin into a file, and is used as a debugging aid only. Macros are used to smooth parameter changes, avoid processor denormalization switching and to restrict output inside system's dynamic range. `MMI` and `MATH` namespaces provide some useful static functions, and `CssString` class extends MFC's string handling with conversion and tokenization functionality.

Chapter

6 Implementation

Armed with the requirements specification and architectural design decisions, it is time to step into the implementation phase. In practice, the process is iterative, though, and sometimes the architecture (and even requirements) do change with issues that emerge while coding. This chapter describes the key algorithms of PHUZOR, and discusses some of the iterative steps that were taken, with a special consideration given to DSP and EH subsystems.

6.1 Core

The interface classes between subsystems have a static method `GetInstance()`, which returns a pointer to the interface's implementer, thus providing a centralized way to access functionality of any subsystem from any place of the code. Corresponding call to `SetInstance()` is made by the implementer during its initialization phase. PHUZOR has also convenience methods in `CoxSynth` class for getting casted subsystem pointers.

VST plugin's entry point `main()` creates `CvstPlug` and `CoxSynth` instances. `CvstPlug` is a simple protocol converter that serves as a router between native plugin architecture and framework classes, and when host signals that performer has loaded the plugin, it calls `CoxSynth::Init()` that in turn initializes `CoxBussManager` and `CoxWavePool` objects, and lets the base class take care of framework's subsystem creation. `CoxSynth` does this by using virtual creation and initialization methods, some of which are overloaded in PHUZOR to provide its custom functionality.

6.2 Event Handling

6.2.1 MIDI Input Stream Processing

Figure 5.3 shows a sequence diagram relating to the MIDI input processing task. PHUZOR initializes itself in response to performer's loading and power on actions, and informs the host that it should start feeding MIDI events whenever they are due. Host obeys, and when live MIDI data is received from external controllers, or when any of the sequencer tracks bound to PHUZOR have MIDI events to be rendered, it dumps those raw events in a `processEvents()` call.

CvstPlug then loops through all events, transforms each native VST event into universal framework format with a call to `InitEvent()`, and passes it through `IFmidiEventX::OnMidiEvent()` for further processing. After all events have been handled this way, a final call to `SortEvents()` might be necessary to put events into timestamped order. At the end of processing cycle, PHUZOR returns 1 to indicate that host should keep on sending MIDI data.

```
long CvstPlug::processEvents(VstEvents* pEvents)
{
    // -- plugin architecture independent structure
    TssMidiEvent ssMidiEvent;

    for (long iEvent = 0; iEvent < pEvents->numEvents; iEvent++)
    {
        VstEvent* pEvent = pEvents->events[iEvent];
        if (pEvent->type == kVstMidiType)
        {
            InitEvent(((VstMidiEvent*)pEvent, ssMidiEvent);
            if (!IFmidiEventX::GetInstance()->OnMidiEvent(ssMidiEvent))
                return 0;    // no more events
        }
    }

    // -- Sort according to deltaFrames
    IFmidiEventX::GetInstance()->SortEvents();

    return 1;    // want more
}
```

Figure 6.1 Processing Midi input data

CssEventHandler class implements the `IFmidiEventX` interface, and the purpose of `OnMidiEvent()` method **A** is to perform filtering of input events, and to store events passing the tests into an internal memory structure for later investigation during audio rate processing cycles. Events are kept in a double-ended queue (ordered by their timestamps and possibly by an explicit `SortEvents()` call), so that it is easy to insert new events into the end of the queue, and to fetch the oldest ones from the front for parsing. STL class `deque` is used for event queue in non-optimized version of PHUZOR.

Filtering algorithm is able to quickly determine whether received midi event is worth of storing, or whether it should be discarded right away. Only events having certain command status and channel pass the filter, and any combination of commands and channels can be configured to be passed using three 16-bit wide masks (one for channel messages `0x8n..0xEn`, another for system messages `0xF0..0xFF`, and one for 16 available MIDI channels). These masks are bitwise *anded* with input event's status byte, and if the result is equal to zero, the event is filtered out.

Ideally, all irrelevant events should be discarded, as it is highly undesirable to burn CPU cycles at audio rate just to notice that a stored event does not have any influence to the produced sound. However, this would complicate input filtering implementation too much, and the benefits gained by ignoring an event altogether might be insignificant, or even negative (ideal approach would require parsing the input data twice).

6.2.2 MIDI Processing at Audio Rate

`IFmidiEventProcessor` interface implemented also by `CssEventHandler` contains two methods which are invoked by the time slicing algorithm of the audio thread. `PeekEvent()` examines the buffered MIDI event queue, and if there are pending events waiting to be processed, fetches the frontmost (i.e. the oldest) event from the queue. It does not remove it from queue yet, in order to make parsing algorithm simpler. `HandleOldestEvent()` is the place where MIDI messages are translated into synthesis commands and parameters. Before this stage, MIDI events were just transformed, filtered, moved around and scheduled, and the decision on what to actually do with the event was postponed to a later date. Processing cannot be delayed anymore, as the time slicing algorithm has, in a sense, made the event current, so this method has to take the responsibility.

`HandleOldestEvent()` **B** pulls the oldest event from the queue, stores its timestamp into an internal variable, and passes the event to `OnParseMidiEvent()` method. It then peeks the next event from queue, and if it has the same timestamp as the first one, it gets the same treatment. This is repeated until the queue becomes empty, or if the timestamp of the frontmost event is later than that of the current event, meaning that its time is due in the (near) future. `OnParseMidiEvent()` method is the top level MIDI event parser, and depending on the status byte of the event, it is sent either to `CssVoiceManager` or `CssControllerManager` for further analysis. All system messages are ignored.

Voice Management

`NoteOn` and `NoteOff` messages are sent to `CssVoiceManager`, which is responsible for voice allocation. The algorithm in the prototype is very basic, as it just ignores `NoteOn` messages if there are no free voices available. In assistance to a more advanced voice allocation algorithm, each voice maintains its internal state (Figure 6.2), a cumulative length in ticks and parsed MIDI information (note number, velocity and channel). It should be noted that voice inactivation is done only after its amplitude envelope has finished its cycle, in order to eliminate audible clicks that would otherwise be present, should the voice be cut off immediately in response to `NoteOff` message.

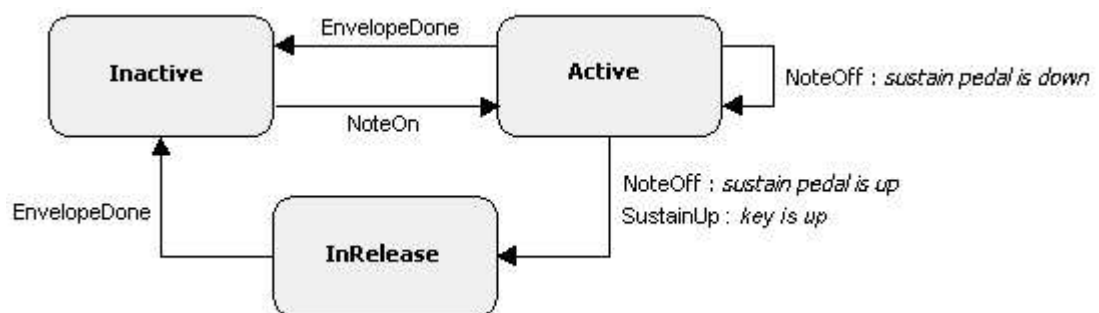


Figure 6.2 Voice states and transitions from state to another.

External Controllers

Controller messages are routed to `CssControllerManager`, which normalizes the data value and stores it into the controller table. The table is scanned at the time of control rate processing cycle. Channel mode messages are parsed separately.

6.2.3 Time Slicing Algorithm

The audio rendering callbacks `process()` and `processReplacing()` are both caught in the `CvstPlug` instance, which combines them into a single `CssSynthEngine::OnProcess()` call, where replacing mode is passed as a boolean parameter. If audio thread was invoked in replacing mode, `OnProcess()` starts by clearing the whole output buffer for silence, because polyphonic implementation assumes that each voice can simply accumulate its audio output to the material already in the output buffer. The time slicing algorithm kicks in thereafter, and splits output buffer (representing a continuous real-time audio stream) into smaller slices, each of which can be synthesized as an atomic entity without being interrupted by control or MIDI rate parameter updates. The algorithm is easiest to explain through an example.

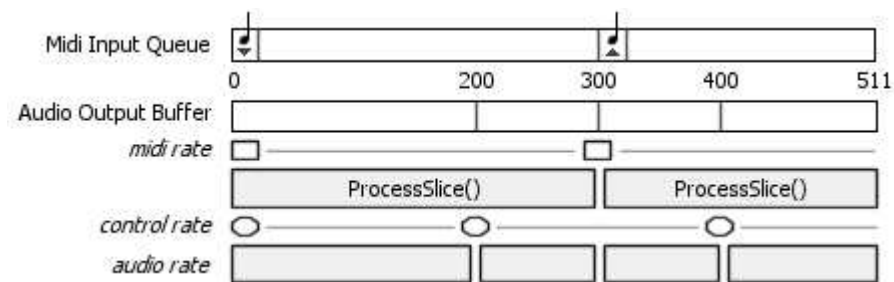


Figure 6.3 Time Slicing Algorithm example.

Let's suppose that the plugin has one monophonic output port, and that the host has a (maximum) output block size of 512 floats. The host then sends a continuous feed of requests to fill that output buffer, and at time $t_0 = 0$ the plugin has already cleared the output buffer. Let's further suppose that at time t_0 there is a single NoteOn message in the MIDI input queue, and a corresponding NoteOff at $t_0 + 300$, and that plugin's control rate is set to 200 ticks (i.e. EGs and LFOs are updated at time t_0 , $t_0 + 200$, $t_0 + 400$ and so on). At time t_0 , `CssEventHandler` is consulted to check the MIDI input queue, and it finds that there is an event to be handled at time offset 0, so it shall parse it, assign a voice to it, initialize dynamic synthesis attributes to their initial state, and finally remove the event from the pending input queue. Next event is peeked, and scheduled to happen at $t_0 + 300$, so output buffer elements [0..299] can be sent to the `ProcessSlice()` method.

`ProcessSlice()` has to make further sub-slicing actions depending on the configured control rate interval. Continuing with the example, the first control rate update takes place at the very beginning of the first slice, so before any audio generation actions can be taken, a global patch level `ProcessControlRate()` method is invoked, followed by corresponding calls to update the modulation parameters of each active voice. After EG

and LFO outputs have all been refreshed and transformed, the output buffer range [0..199] can be synthesized. Internal control rate update counter is then increased, and at $t = 200$, a new control rate refresh cycle is initiated. Finally, the output buffer range [200..299] is synthesized.

Once the first slice has been filled with synthesized output values, internal time counter is increased, so that it now points to time offset 300. Event handler's parser is invoked to manage the NoteOff message, which eventually empties the midi input queue, leaving slice [300..511] to be sent to `ProcessSlice()` method, which has still to sub-slice it at time $t + 400$ in order to perform control rate refreshing. After slice #2 has been entirely processed, the whole output buffer has been filled, and can be returned to the host for further processing and audio output. Shortly a new buffer fill request will come in, and the whole process is repeated with possible MIDI input queue handling, and control rate refreshes occurring at times $t + 88$, $t + 288$ and $t + 488$.

6.3 DSP

6.3.1 Oscillators

Table Lookup Oscillators

PHUZOR oscillators have gone through couple of revisions before they have matured into the versions that are used in the prototype. The first revision used ideal mathematical representations of the waveforms, e.g. a square wave was generated by setting output value to 1.0 when instantaneous phase was less than half the wavelength, and to -1.0 when it was advanced into the second half. Of course, this algorithm aliased heavily, and another solution was necessary. One could use over-sampling in order to shift aliasing frequencies upwards in frequency scale, and then low pass filter the decimated signal, but it was decided to use additive synthesis techniques instead. The second revision oscillators were implemented as straight-ahead sinusoidal oscillator clusters. The results were promising, as there was no audible aliasing, but there was still a problem because it was far too CPU intensive to be practical. So finally it was decided to use precalculated table lookup oscillators for stock waves, user spectra and naturally for sampled waveforms as well.

The inner loop of basic table lookup oscillator algorithm of PHUZOR is shown in Figure 6.4, which illustrates that per sample calculation and conditional branches are kept at absolute minimum, and that all processing is kept inline. The algorithm uses linear interpolation, because the improved signal-to-noise ratio is worth the extra CPU cycles (figures in Table 2.1 were evaluated in practice, as early versions of PHUZOR utilized truncating and rounding alternatives, and the results proved the figures correct). It was decided to use 1024 byte table sizes instead of 512 in order to achieve 108 dB SNR, because with 32-bit IEEE floats will give 25 bits resolution, and consequently it is possible to reach at least 150 dB (sign bit + 24 bit mantissa, $20 \cdot \log(2^{25})$). The number of wavetables per oscillator is 1 for sine waves (implemented as `CoscTableLookup`), and 128 for triangle, square and sawtooth waves (`CoscMultiTableLookup`) and pulse waves (`CoscVariablePulse`), i.e. there is a separate table for each MIDI key, holding entire

range of a standard concert grand (88 keys) plus an octave and a fifth. Pulse waves with variable width duty cycles are generated in real-time by superpositioning an inverted and phase shifted sawtooth wave with another positive polarity sawtooth. Different pulse widths are then achieved by modifying the phase shift amount.

```

while (sliceLength > 0)                                // for entire block of samples
{
    // -- table indexes
    fPhase = fmod(m_phase, iTableLength);
    iTableIndex = int(fPhase);                          // integer part
    fFract = fPhase - float(iTableIndex);              // fractional part
    if (iTableIndex < iMaxIndex)
        iNextIndex = iTableIndex + 1;
    else iNextIndex = 0;

    // -- linear interpolation
    fSample1 = m_pWaveTable[iTableIndex];
    fSample2 = m_pWaveTable[iNextIndex];
    fValue = fSample1 * (1.0 - fFract) + fSample2 * fFract;

    // -- store output
    *pfOutputBuf++ = fValue * fAmp;

    // -- interpolate control rate modulation destinations
    fAmp += fAmpDelta;
    fPhaseIncrement += fPhaseIncrementDelta;

    // -- prepare for next sample
    m_phase += fPhaseIncrement;
    sliceLength--;
}

```

Figure 6.4. Table lookup oscillator algorithm.

The wavetables are created at initialization time using IDFT. Based on current tuning of concert A and 12-tone equal temperament, the number of non-aliasing harmonics is calculated, and if this number is equal for two consecutive keys, then a single wavetable is shared by both keys. As it turns out however, each key has different amount of non-aliasing harmonics, so a separate table is assigned to each key. Thereafter, harmonics up to the calculated number are summed together according to Fourier series of waveform, or up to the point where harmonic is considered to have no impact on perceived sound (this happens when its amplitude is below certain threshold value, which is currently set to -108 dB below normalized fundamental component amplitude, see SNR above). This process is repeated for each sample so that full cycle of the waveform is formed.

The playback algorithm of `CoscMultiTableLookup` is identical to that shown in Figure 6.4, with the difference that the wavetable that is being scanned is determined by a combination of current key number, pitch parameter settings and possibly by a pitch modulation value. If the pitch is being modulated, instantaneous frequency is quantized to a base frequency of the tuning table, and nearest matching wavetable is used for scanning. The fractional deviation from the base frequency is used as an offset when determining the sampling increment. As the modulation value is likely to change in course of a note event, scanned wavetable might be changed to another at control rate, so that nearest precalculated wavetable is always used as the base.

Spectral definition waves loaded from SPE files are precalculated using the same principle, and played back using `CoscMultiTableLookup` oscillators, because single

file can contain definitions for up to 128 key ranges. The prototype has no velocity zones, so it is impossible to have different source material assigned to key velocity.

`CoscSample` uses likewise the multitable approach. In addition to the basic table lookup algorithm, it has means to manage looping points inside the main synthesis code. After each `m_phase` incrementation, phase value is compared against loop end point, and if it is beyond that, the loop region length is subtracted from `m_phase`. Backwards looping is handled similarly, but this time `m_phase` is *decremented* by sampling increment, loop *start* position is examined and region length is *added* instead of subtracting. There is no crossfading implemented at loop points in the prototype.

An interesting supplement to the standard table lookup oscillator algorithm is the use of pseudo oscillators, detuned slightly sharp and flat in respect to the actual oscillator's pitch. Each pseudo oscillator uses a unique set of `m_phase` and `m_phaseIncrement` variables, which are used to calculate separate output value for each pseudo oscillator. All output values are finally added together and scaled by the number of pseudo oscillators in use, and outputted as the value of the entire oscillator.

Algorithmic Oscillators

By definition, the noise oscillator cannot use precalculated values. Its output must be generated for each sample algorithmically, and should thus be as efficient as possible. The algorithm that is used in PHUZOR is linear congruential generator [54], which uses a recurrence relation and has just single multiply and add, plus a floating point divide for range scaling. It is fast.

The Bézier oscillator `CoscBezier` used in the waveshaper particle is more problematic, as solving time from Equation 2.8 is quite tedious. This was also noticed by Lang [18], and he suggests using Newton-Raphson iterative method for root finding [54], which is used also in PHUZOR. Evaluation has to be performed for each sample, but the oscillator retains the previously computed root in its state variables and uses that as the initial guess in the algorithm. The epsilon to stop the iteration is set to 10^{-6} , which yields just about 2.2 iterations on the average.

The plucked string oscillator `CoscPluck` can use noise or any of the precalculated or loaded waveforms as the excitation source. After initial excitation, the delay buffer is lowpass filtered using 3-point averaging FIR filter *squish* parameter times (modeling pick material), and with one-pole lowpass usually modulated by key velocity. The initial amplitude of the fill material can also be determined by key velocity. Delay line is implemented as a simple ring buffer with one combined read/write pointer (this means that oldest value is read from the delay line before new value is written, after which the single pointer is incremented modulo ring buffer length). Value from delay line is copied into output buffer, and put back into delay line via 2-point averaging lowpass and allpass filter to provide fractional delay line lengths. Decay damping and stretching settings are taken into account when computing the output value.

6.3.2 Modifiers

Container

`CoxModifierBlock` class has two `CssModifier` instance member variables, which can be dynamically created when patching the sound, or attached to the pre-constructed instances when loading a patch from a file. It serves as a parameter, control rate modulation, and audio rate rendering router between actual modifiers and other parts of the framework. The most important algorithm of `CoxModifierBlock` are the various `Process()` methods, optimized for each possible port configuration (i.e. mono-to-mono, mono-to-stereo and true stereo modes). Inside these methods, output buffers are passed to the individual modifiers according to current modifier topology (serial or parallel), taking care of not cutting the signal flow in case one or both of the modulators are set to bypass state.

Modulation Effects

`CfxPhaser` is implemented using up to 12 first order allpass filters connected in series. Each sample is fed through an allpass filter parametrizable *stages* times. The filters have been tuned in octave steps, so at maximum there are 6 notches in the composite filter spectrum, located at multiples of modifier's *frequency* parameter. The allpass delay values are modulated at control rate with an LFO (for stereo versions with two LFOs having 2nd polarity reversed). The LFOs are by default inside the `CfxPhaser` class, but it is also possible to use an external modulation matrix source for them. This facilitates per channel rate, waveform, delay, amount and phase settings.

`CfxChorus` is used for both chorusing and flanging effects, and it is implemented using two variable fractional length delay lines in parallel, with an independent internal LFO for each channel (for monophonic configuration, there's only one of each). Delay lines are ring buffers with combined read and write index, and a fractional delay offset that is calculated at the start of the audio rendering cycle. The delay line contents are updated from the input stream, and from the output value fed back for flanger effects.

Spatial Effects

`CfxFixedDelay` uses a fixed length ring buffer as a delay line, where fixed means that although the length is parametrized, it cannot be modulated at a course of a note. The buffer is described with a start pointer to allocated memory block, and with a size of that block, defining also the maximum achievable delay time. For convenience, there is also an end pointer that points to the end of the allocated block. Current write position to the buffer is also kept in a pointer, which traverses from start pointer to the end pointer, after which it folds back to the start of the buffer.

When *delay time* is changed by the user, read pointer is offset from the write pointer with an amount equal to the set delay time in samples. During audio rate rendering, output sample is fetched from the read pointer location, mixed with the incoming sample according to *dry/wet* parameter, and stored into the output buffer. Incoming sample is combined with the sample that was read from the delay line according to

feedback parameter setting, and written into the delay line at the write pointer location. Finally, read and write pointers are increased within buffer boundaries.

There is also a stereophonic version of `CfxFixedDelay`, which has two delay lines running in parallel. It contains *crossfeed* parameters defining the amount of to-be-delayed signal that leaks into the other channel. This facilitates complex rhythmic delay effects, which can be further animated using channel specific panning.

`CfxFreeverb` is based on a public domain Schroeder/Moorer reverb implementation [55] with eight parallel comb filters per channel, the output of which are accumulated together and fed through four allpass filters in series. This version does not have control over predelay or filtering parameters.

`CfxSteroid` is a stereo image width enhancer. It operates on L-R difference signal by feeding it through an internal delay line similar to that of `CfxChorus`, and adds the processed signal to original left channel, and an inverted version to the right channel, with parametrized gain amount. There is also a pseudo stereo generator for monophonic input implemented using same basic principles.

Filters

All filter classes are derived from an abstract `CFilter`, each implementing two basic methods. `CalcCoefficients()` is called at control rate each time filter's frequency or resonance parameter is changed (by performer or by modulator). Coefficients are stored internally, and used at audio rate rendering `Process()` methods, which take input signal and internal state members stored from previous inputs and outputs as operands. Output value is computed from these operands and coefficients with a dedicated algorithm for each `CFilter` subclass.

There are numerous filter implementations in PHUZOR, most of which are straight forward IIR incarnations of 6, 12 and 24 dB / octave designs. Of particular interest are the four pole lowpass Moog ladder filter [8] emulations based on algorithms by Stilson and Smith [56] and Huovilainen [57]. The latter uses a cascaded tanh network, two point averaging FIR for phase compensation, two times oversampling because of the nonlinearities, and a lookup table for performance boosted tanh evaluation.

Other Modifiers

`CfxBlender` is a two channel stream processor that is able to extract or generate left, right, sum and difference signals from a stereophonic source, invert the extracted sound and pan it independently. It facilitates for example MS pair encoding and decoding, L-R signal inversion, crude stereo-to-mono mixing and channel swapping. The algorithm operates on single sample only (i.e. it does not have any internal state variables), and is consequently very simple and effective, allowing it to be used in series with other modifiers without any noticeable CPU overhead.

`CfxWaveshaper` drives input stream of samples through a 1024 point transfer function, which has been scaled to cover the range from -1.0 to 1.0. The input sample defines the

shaper table index, and the output of the modifier is transfer function's value at that index (and the next index, using linear interpolation).

6.3.3 Containers

`CoxSourceSection` contains members for all eight `CpartParticle` instances, and its main purpose is to act as voice gating, parameter change notification, control rate modulation, and audio rate rendering router between particles and other parts of the framework. Its `Process()` algorithm optimizes the use of busses so that unnecessary clearing and copying does not slow down synthesis algorithms. Processing triggers also the ARM synthesis methods and provides proper streams for participating oscillator inputs by mixing modulating audio streams together, scaled by the square root of the number of input streams (it would also have been possible to use more conservative $1/N$ scaling, but as signals are usually uncorrelated, i.e. there are no peaks at the same time, clipping should be consequently rare). See also section 6.3.5.

`CpartParticle` and its descendants delegate much of their work to `CoscOscillators` and `CoxModifierBlocks`. They do handle the panning inside their `Process()` methods, and for that reason implement also `ProcessControlRate()` method. `CpartWaveshaper` class is more complicated, as it implements the actual transfer function lookup actions after source oscillator has been asked to provide the needed indices. The lookup algorithm is similar to that of `CfxWaveshaper` class.

`CoxMixerSection::Process()` methods route the input stream first through the modifier block, and perform the necessary mixing and panning calculations into the send and main mixer busses inside two tight loops thereafter. `CoxMasterSection::Process()` first scales input signals by dividing them with a square root of the number of active channels -scheme (see above), then routes the scaled signals via modifiers, and finally mixes and pans all three channels together into a stereo pair. This master signal is then routed via mastering modifiers (should there be any in use), and to complete the synthesis process, copied into audio output buffer with parametrized panning and level scaling.

For safety reasons, master output is run through a `restrict` macro, which clips all out of range samples into the host's $[-1.0..1.0]$ range. Before clipping, there is also an audio fuse, which mutes the output if predefined number of out of range samples is detected during single output block processing.

6.3.4 Control Rate Modulation

Modulation Matrix

Control rate update cycle is managed by `CssModManager` class, which iterates through all active cords in the control rate modulation matrix, ticks each source to get current modulator value, and stores that value into the cord. After all cord sources have been ticked, active cords are iterated once again, this time to check if any of the active cord

destinations is the *amount* parameter of another cord. In such cases, destination's amount value is updated according to source's current value. This iteration is a one-pass process, so cords can be chained only if cord destinations are forward references. All feedback connections are activated during next control rate update cycle. Finally, the cord array is iterated the third time — this time active cords are grouped by destination, and all values from cords connected to a single destination are accumulated together (after multiplying each with cord's amount), and composite value (with possible curve transformation) is transferred into destination's modField.

Sources

As modulators are ticked at control rate, and aliasing is not relevant in these cases, their implementation is simpler than that of audio rate signal generators. Instances derived from `CssModulator` maintain their current time domain position in `m_phase`, which is either reset to zero when modulator is gated (in response to a note on message), set to a random value for LFOs oscillating in freerun mode, or to a predefined initial phase.

Inside each `Tick()`, `m_phase` is incremented with a delta time that defines the interval between two consecutive `Tick()` calls, and if `m_phase` exceeds current segment/cycle length, either new EG segment values are transferred into current values, or `m_phase` is reset to the beginning of a new cycle.

EG segment's state variables consist of begin and end levels, length of current segment (in control rate ticks), and segment's slope. If slope is set to 0.5, EG segment is linear, and current level value is determined from start and end levels using simple linear interpolation. On the other hand, if slope is larger or smaller than 0.5, segment is either exponential or inverse exponential, and current value is determined from equation

$$y(x) = y_1 + (y_2 - y_1) \frac{1 - \exp(\alpha T(x))}{1 - \exp(\alpha)}, \quad T(x) = \frac{x - x_1}{x_2 - x_1} \quad (6.1)$$

where y_1 and y_2 are start and end levels, x is the time in ticks, and α segment curvature calculated from slope of range [0..1] as $\alpha = 20 - 40 * \text{slope}$. The equation is from `cmusicgen4` routine [58], and the slope has been scaled into common VST parameter range.

LFOs calculate their current value either via table lookup, using pseudo random number generator, or algorithmically in case of pulse waveforms. RVGs get ticked when source MIDI event is received, at each control rate cycle, or when source LFO's pulse transits from positive value to negative, depending on RVG's trigger mode. It first increments an internal counter and compares it to the patched *counter* parameter, and if they are equal, a random value is generated and set as current value. If trigger mode is MIDI event, the received data value is compared to patched *threshold* parameter, and new value is drawn accordingly.

MIDI controller data values are stored in a controller table, which is updated by `CssControllerManager` each time an external controller message is received via MIDI.

Ticking external controller sources is just a simple table lookup operation. Widgets have a `GetValue()` method which is called during main control rate update cycle.

Destinations

Oscillators and other destination parameter containers are also ticked at control rate. Inside `OnProcessControlRate()` method, a target value is computed from parameter's current value and its corresponding `modValue` field, giving the value that is reached at the end of the audio rate rendering cycle. A delta value that is added on single sample interval can then be calculated, and this delta value is used inside `Process()` method to linearly interpolate destination synthesis parameter at audio rate, eliminating zipper noise entirely. Finally, current synthesis parameter value is updated so that is ready to be modulated during next control and audio rate processing cycles.

6.3.5 Audio Rate Modulation

PHUZOR particles comprised initially two oscillators that could be internally connected to form a modulator-carrier pair. Carrier oscillator could also be externally modulated by another particle, and the ARM matrix held such connections for four particles. Although this scheme enabled fast patching of certain timbres (e.g. detuned two-oscillator analog-like sounds), it restricted other patches (e.g. DX7-style 6 operator setups), and was later replaced with a more flexible architecture that allows any oscillator to operate either as a carrier or as a modulator, or as both at the same time.

Audio rate modulation is handled within `CoxSourceSection::Process()` method, which is called once per output block for each active voice. The algorithm iterates through all active particles, and if there are other particles modulating current particle, outputs of those particles are mixed into a temporary buss, which is then given as the input stream for the `CpartParticle::Process()` method. As only those particles that have smaller index (i.e. particles inside `CoxSourceSection` are contained in an array, indexed from 0 to 7) can modulate current particle, the output streams of modulators are already synthesized and stored inside the source buss. This restriction emerges from block-oriented processing architecture, and there would naturally be no such constraints if synthesis was done on single sample basis.

The lack of feedback connection between particles is a consequence of this restriction. There is a feedback path from particle's output into the input of the same particle, though, and it is handled inside the particle's `Process()` method. It should be noted that the only ARM type that must be handled inside oscillator's synthesis loop is *FM*, as it affects the phase increment of the table lookup oscillator¹. *Pass* ARM type is reduced to modulated block copy operation without the need to synthesize carrier output at all.

¹ In PM current sample is read from a lookup table before modulation is applied to the lookup index, while in FM the lookup index is updated prior fetching. PHUZOR does actually PM.

Sync is also a special case, where input stream data is ignored and only modulator's frequency is of importance. There might also be control rate modulation applied to the modulating stream (most commonly using EGs), resulting one additional multiplication in the algorithms.

Figure 6.5 shows a simplified version of PHUZOR's XOR audio rate modulation algorithm, and the corresponding SSE optimized code in Figure 6.6. Control rate modulation code has been omitted for clarity. As can be seen from the optimized version, there shall be a remarkable speed improvement, because FPU can compute XOR operation using single instruction and as it can calculate four samples in parallel.

```
while (sliceLength-- > 0)
{
    // -- get input values and switch to 16-bit integer processing
    fCarrier = *pfStreamCarrier++;
    fModulator = *pfStreamModulator++;
    iCarrier = int(fCarrier * 65535);
    iModulator = int(fModulator * 65535);

    // -- compute and store output
    iXOR = iCarrier ^ iModulator;
    *pfOutputBuf++ = float(iXOR) / 65535.0;
}
```

Figure 6.5 Audio rate modulation example. Simplified C++ implementation.

```
int iPackLength = iStreamLength / 4;
for (int iPack = 0; iPack < iPackLength; iPack++)
{
    // -- compute and store output
    *pm128Dest = _mm_xor_ps(pm128Modulator, pm128Carrier);

    // -- prepare for next pack
    pm128Dest++;
    pm128Modulator++;
    pm128Carrier++;
}
```

Figure 6.6 SSE instruction set optimization of previous example.

6.4 MMI

PHUZOR's MMI is implemented using a combination of subclassed Windows common controls and custom controls developed for software synthesizer specific interaction tasks. Implementation is based on straightforward adaptation of model-view-controller paradigm of MFC and its `CFormView` class, where pages and panels comprising the MMI are designed using a graphical resource editor provided with the development tools. Specialized widgets include graphical envelope editors, multifield edit controls with a virtual slider (where vertical mouse dragging operation is used to alter numerical field values), four-state checkboxes and menu buttons fusing a combobox and a popup menu into a single control.

Of particular interest is the Structure page, which is used to interface PHUZOR's audio rate signals (Figure 6.7). It combines the functionality of Line and Master Mixer pages with the audio rate modulation matrix into a common surface, that facilitates signal routing between particles, modifiers and mixers, output and send level settings, panning,

and switching of particles and mixer channels. It allows also configuration of audio rate modulation type between particles.

There are nine diagonally positioned outlined particle boxes (the leftmost being the external audio input), and four outlined master mixer channel boxes at the bottom right of the page. ARM matrix and line mixer controls are located between outlined boxes, where each node represents a connection between the source (the outlined box vertically above the node) and the destination (the outlined box to the right of the node). Node is dimmed if there is no connection. A connection is made by left clicking the node, and by dragging mouse vertically up and down in order to set the output level of the source. Once connected, the node is visualized with a yellow number inside a solid box, and lines connecting the source with the destination are drawn to indicate the participants. The three bottom most node rows contain two nodes per connection, where the right one is used for pan position definition.

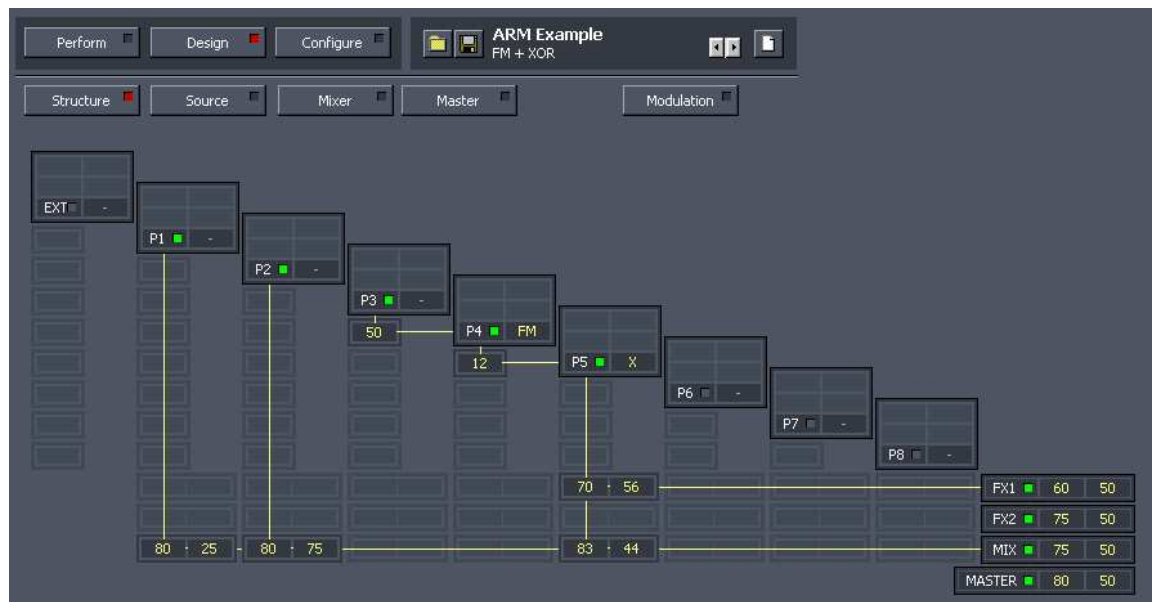


Figure 6.7. Structure Page. Particles 1,2 and 5 are carriers, while 3 and 4 operate as modulators (FM and XOR, respectively).

Particle boxes are further assembled into six internal boxes, which are used to define the particle state (on/off), ARM method, self modulation amount, ARM input level setting, and particle modifier send level settings. Master mixer boxes contain 3 internal boxes for on/off switching, and output level and pan settings.

6.5 Parameters, Patches and Files

Parameters and Patches

`CoxPatch` groups all synthesis parameters under a single accessible unit, breaking them into structures by controlled element. The class is also able to take a parsed parameter file as input, and transfer values into places that are accessible from the synthesis elements.

`CoxParameterHandler` contains a common method to set a particular parameter value of the current patch, and has knowledge to notify parameter containers when changes have been made. The implementation is a simple parser basically in form of a large switch statement, whereby case labels are parsed from parameter's scope and id, as provided by the method caller.

Patch and Subpatch Files

Each persistent object implements `IFPersistent` interface's `OnLoad()` and `OnStore()` methods, which stream object's attributes from/to disk files. Attributes are stored in human readable form, and are at the lowest level accessed using Windows profile API calls. `CoxPatchFile` and its descendants buffer file input via persistent object members, and after successful parsing and parameter loading, the MMI passes file instance to the `CoxPatch` class, which then transfers buffered input to the current patch. To accomplish this, each persistent object overloads the assignment operator, which copies attributes from source instance to the destination. Storing is not buffered. At the start of the store operation, file is truncated to zero length, after which a section header is written, and each persistent object belonging to the container section then writes itself to the file.

Other Files

Waveform files are for input only, and are parsed and loaded through an internal buffer into `CoxWavePool`'s structures. Oscillator wavetable pointers and other patch parameters are updated accordingly.

Spectral definition files are also for input only, and handled in a similar way as waveform files. `CoxWavePool` builds a set of wavetables out of the definitions using the Fourier summation algorithm as described in section 6.3.1. The number of wavetables that are created depend on the number of groups defined inside the SPE file.

Waveshaper transfer function definitions, i.e. the degree and the polynomial coefficients are passed after parsing once again to the `CoxWavePool` class, which calculates the transfer function for values between $[-1..1]$, and stores the generated wavetable in internal structures for use inside `CpartWaveshaper` and `CfxWaveshaper` instances.

6.6 Settings and Utilities

`CoxSettings` contains PHUZOR's configuration setup for such items as directory paths of patch and sample files, MMI skin definitions, master output level and pan position, master tune and the number of voices. It is derived from `CwinRegistry`, which interfaces Windows registry API. Values are read from registry at initialization time, and are written back whenever modified within *Configure* page. All configurable items are static and public, thus easily accessible from any part of PHUZOR.

`CssString` which is derived from MFC `CString` class contains a handy implementation of a simple parser. It is initialized with a call to `InitTokenizer()`, which sets the separator character and resets internal counters. It operates on the contents of the string

itself, and each call to `GetNextToken()` returns the substring that is delimited by separators, or zero if there are no more tokens. It is used extensively when parsing patch files, as are the conversion methods `AsInteger()`, `AsFloat()` and `AsBoolean()`.

`CssLogger` writes each message between host - plugin communication as a formatted ASCII string, which is useful when debugging specific host incompatibility problems. It can be used for other debugging purposes as well, because breakpoint based debugging might be impossible in real-time.

Utilities subsystem contains also macros for inlined support code. One of the macros is used to remedy the denormalization switching problem of Pentium processors, which manifests itself as a huge CPU overhead when floating point calculations operate on very small magnitudes (according to Intel, the factor can be as much as 250:1). The origin of the problem lies in IEEE floating point specification, which states that very small numbers must be treated in a special way, instead of truncating them to zero. The macro works by adding a small, but sufficiently large number to its operand so that switching is not made.

Chapter

7 Evaluation of Results

This chapter describes the measurements that were conducted on the PHUZOR prototype. First, four different algorithms for sawtooth oscillators available in PHUZOR are compared with each other, and to that of a hardware vintage analog synthesizer. Second, a six oscillator FM patch is programmed, and the sound of PHUZOR is confronted with the sound of a commercial software synthesizer using an identical patch, and accordance with the FM theory is discussed. Third, time and CPU related performance measures are taken, and finally, some quantitative metrics characterising the implementation phase are calculated.

7.1 Sound Analysis

Time and frequency domain graphs and spectrograms were used for sound analysis. The sound material was first captured into a 16 bit, 44.1 kHz WAV file directly from the PHUZOR's outputs, so that sound card unidealities did not interfere with the results. The file was then loaded into MATLAB [59] and ran through a custom script to produce the plots shown below. FFT was averaged from the entire sample using a rectangular window because of the periodicity of the sources.

7.1.1 Sawtooth Waveforms

PHUZOR Stock Sawtooth

PHUZOR was patched so that only one wavetable particle was active. The IDFT generated sawtooth was used as the waveform, and EG was constructed to be a simple gating curve. Two notes (middle C at 261 Hz, and a C one octave below that, i.e. at 130.5 Hz) were played.

As can be seen from Figure 7.1, there is no aliasing, and both waveforms contain a full set of harmonics. There is a small overshoot in the amplitude inherent in the time domain representation, due to the uncompensated Gibbs effect.

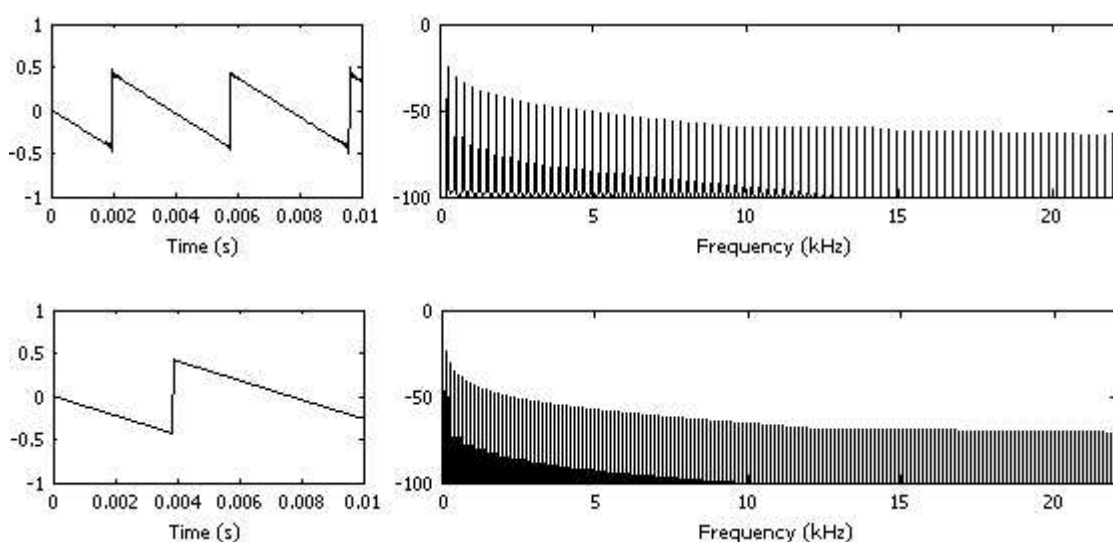


Figure 7.1. PHUZOR stock sawtooth waves. Thick lines at the bottom part of the spectrum plots do not indicate audio aliasing, but are due to graphical imperfections.

Sampled Minimoog

A Minimoog sawtooth sample was downloaded from the Internet [60]. The sample was a one-second raw unfiltered sawtooth with fundamental frequency of 261 Hz (i.e. middle C), recorded at 44.1 kHz and with 16 bit resolution. The sample was loaded into PHUZOR's wavetable particle and middle C was pressed.

The downloaded sample included aliasing, which can be seen from the spectrum plot just below 20 kHz and up to the Nyquist frequency. The interesting bit is the time domain graph showing exponential like ramp.

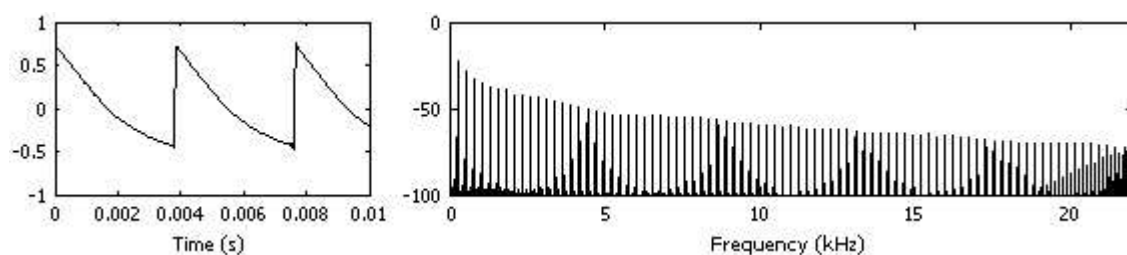


Figure 7.2. Sampled Minimoog wave. Here we have also audio aliasing.

Spectral Reincarnation

For evaluation of the SPE format waveforms, the original Moog sample was loaded into Matlab, and extracted spectral peaks were exported via clipboard to Excel spreadsheet. The amplitudes were transformed into floating point range [0..1], and partial frequencies were regenerated to match the ideal sawtooth (Moog harmonics were in general about 6 cents flat). The file was loaded into PHUZOR and once again, middle C was pressed. The results can be seen in Figure 7.3.

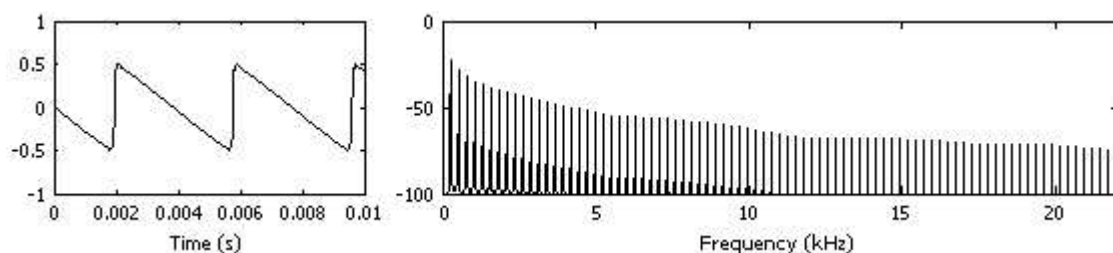


Figure 7.3. PHUZOR's spectral version of the Minimoog sample. Only amplitudes were extracted from the original sample.

Aliasing has disappeared, and the higher harmonics have lower strength than those of the theoretical sawtooth of Figure 7.1. The time domain graph of the waveform is not identical to that of the sampled version, however, as partials' frequency and phase information was not used in resynthesis. Unfortunately it is not possible to create inharmonic spectra without aliasing. It is nevertheless possible to use the exact (within FFT resolution) partial frequencies if aliasing does not sound too bad, and the produced timbre is naturally more close to the original than the stripped version, which sounds somewhat duller than the original.

Supersaw

Figure 7.4 shows the waveform and spectrum of PHUZOR's supersaw waveform, with six pseudo oscillators detuned 7, 14 and 21 cents flat, and 4, 8 and 12 cents sharp in relation to the fundamental pitch. Pseudo oscillators have 80 % of the amplitude of the center oscillator.

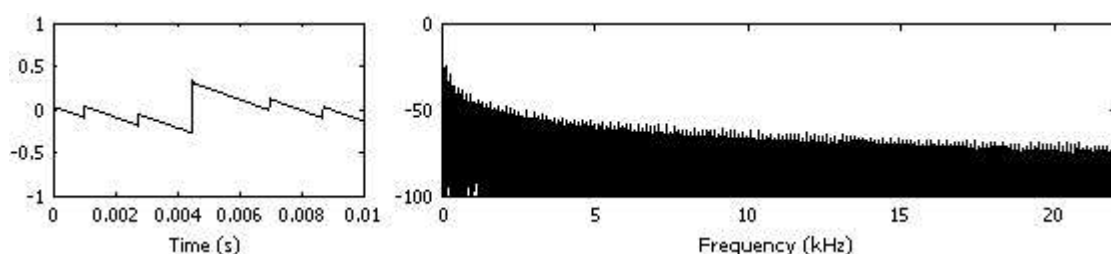


Figure 7.4. PHUZOR supersaw played at C3.

The sound is fat indeed, and is the output of only one particle. With eight particles, each detuned slightly from the others, and panned throughout the stereo space, the output starts to resemble the sound of a tuned noise (this would equal 48 sawtooth oscillators in unison).

7.1.2 FM-style Electric Piano

Patch Description

Probably the most characteristic sound often associated with Yamaha DX7 synthesizers is the digital Rhodes emulation, or the factory patch *FullTines* of the mark II range. It uses three parallel two operator stacks — two of the stacks giving the body of the sound (i.e. the rubbery sustain part, slightly detuned and panned around the center), and one

that synthesizes the bright attack portion (emulating the striking of the metal bars of Rhodes electric piano). The modulator - carrier frequencies of the sustaining operators are of ratio 1:1, and thus produce a sawtooth like mellow timbre. The attack portion on the other hand produces inharmonic metallic overtones because of the 12:1 modulator - carrier ratio, where the modulation index changes from full range to zero in about 2 s.

FM7

Native Instrument's FM7 [61] is capable of accurate DX7 emulation, and is even compatible with its MIDI sysex dump format. The original *FullTines* factory patch was imported into the FM7, and a sequencer was used to trigger a full velocity, six seconds long note at frequency $f_0 = 261$ Hz. The spectrogram of Figure 7.5a clearly shows the sustaining body of the sound (all harmonics up to about 2.5 kHz), and the output of the 12:1 tuned operator stack as two peaks occurring at $k*12f_0 \pm f_0$ which constitute to the attack portion and decay rather rapidly.

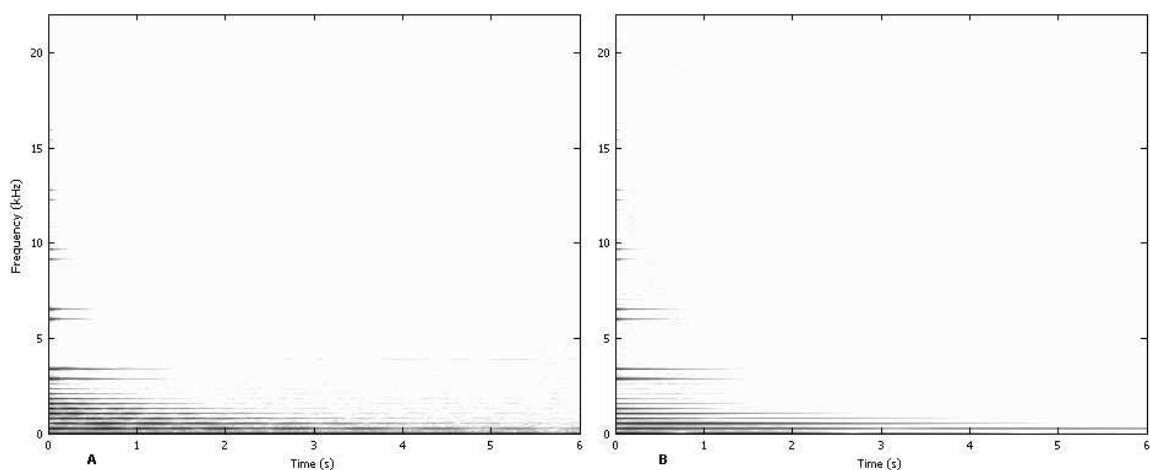


Figure 7.5. A) FM7 output of *FullTines* patch. B) *PHUZOR* version.

PHUZOR's Emulation

Envelope times, levels and slopes were manually transferred from FM7 into PHUZOR, and six wavetable particles with sine wave sources were then structured into a similar ARM matrix as the original DX7 algorithm 5. Particle output levels were also set to the same values that were in the original *FullTines* patch. Other parameters of the original patch were discarded.

The spectrogram in Figure 7.5b is nearly identical to that of FM7. The modulation index mapping into modulator output level is not exact, as can be seen from smaller number of harmonics below the two peaks at 2.5 kHz, but it is very close. The slightly wavy pattern of partials in FM7 output is due to the detune parameters, which were not set in PHUZOR patch. Minute partial length differences are due to unequal EG implementations.

7.2 Performance Measurements

Performance related timing figures were measured using Win32 API high resolution performance counter functions, which give at least 1 μ s resolution within the testing environment. In addition, the CPU load meter values from MiniHost application were recorded. The metric is presented as a percentage value, ranging from 0.3 % when idling to full 100 % when the entire time scheduled for the process is consumed. These figures follow quite closely the measured timing values, suggesting similar derivation.

The output buffer size of 512 samples per channel was used in measurements. Only single notes were triggered, and each measurement were repeated ten times to get the listed average. The fourth column of Table 7.1 contains the maximum polyphony achievable if all CPU cycles were used to synthesize the timbre shown in the first column. These values were calculated from the time required for synthesis of single voice, column 2), and from the amount of time that is available per block of drop-free audio streaming (at 44.1 kHz sample rate with 2 output channels equalling 5.805 ms). The release configuration of PHUZOR was compiled with speed optimized, but without SSE tuning.

Table 7.1. PHUZOR performance. Time is in milliseconds, unless otherwise noted.

General	Time (ms)		
Startup time	10..11 s		
MIDI input and MMI update			
MIDI in -> event queue	0.023		
MIDI in -> synthesis start (i.e. latency - synthesis time)	0.056		
Oscillator pitch doubled from MMI -> sound output	7.816		
Synthesis of one output buffer		CPU (%)	Polyphony
1 wavetable particle with stock sawtooth wave	0.062	0.7	94
1 waveshaper particle with Bézier wave	0.120	1.3	48
1 pluck particle	0.075	0.8	77
1 wavetable particle with supersaw	0.143	1.3	41
supersaw synthesized from 7 detuned stock sawtooth waves	0.171	1.6	34
6 - particle FM patch	0.361	2.6	16
Synthesis of one output buffer using Modifiers			
1 filtered (24 dB LPF) wavetable particle with sawtooth	0.110	1.1	53
1 wavetable saw particle run through chorus (insert)	0.148	1.4	39
1 wavetable saw particle run through master reverb (send)	0.296	2.5	20

The results of the performance tests were encouraging, as the goals set in requirements specification phase were clearly fulfilled. The total latency time, from the moment that MIDI input is available in PHUZOR's input port to the point when synthesized output buffer is ready, is naturally dependent on the complexity of the patch, but with the timbres listed above, only the FM patch and reverbed sawtooth exceeded the latency time of one sample. Reverberation is a rather demanding effect, as expected, but when used as a send effect, the CPU requirements stay constant regardless of amount of polyphony. On the other hand, the FM algorithm and the initial startup time need further optimization. It is safe to assume though, that SSE code at selected spots will improve the figures above with a considerable amount, especially those associated with audio rate modulation.

7.3 Source Code Analysis

Quantitative source code analysis was made using SourceMonitor [62], which is an off-line tool that is able to parse C++ code and calculate a collection of metrics from the results. One of such metrics is the number of C++ classes. Including struct definitions, the prototype version of PHUZOR comprises 188 classes, of which 63 (34 %) belongs to the framework package, and 125 (66 %) to the dedicated synthesizer implementation. Some of the classes of the latter package, especially those implementing such standard elements as table lookup oscillators and filters, are also quite reusable, so they could be eventually transferred into the framework domain.

Another metric is the number of statements (i.e. the lines terminated with a semicolon character and control structures), giving a better indicator of the size of the code than pure number of lines (which includes comments and empty lines as well). The total amount of statements is 19160, which distributes to 5191 (27 %) and 13969 (73 %) instances between packages. Figure 7.6 shows how PHUZOR statements are scattered amongst the subsystems of Figure 5.2.

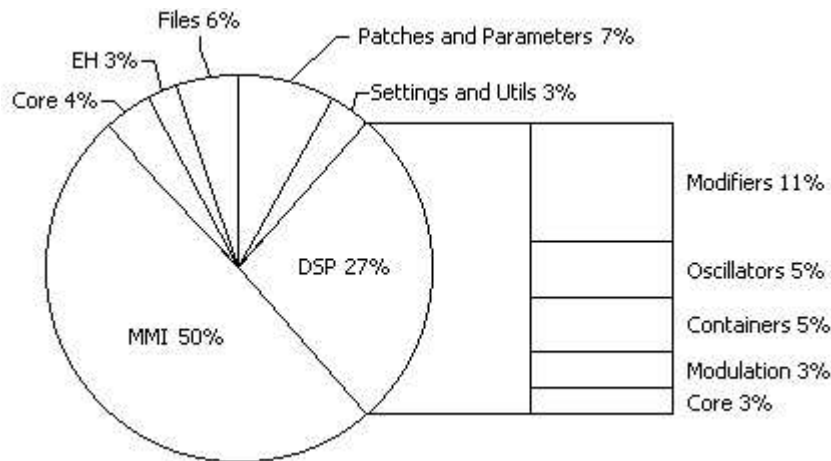


Figure 7.6. Functional distribution of PHUZOR source code statements.

The most striking aspect of Figure 7.6 is the amount of MMI code, which is half the entire amount, and nearly twice as much as the amount of DSP code. Although the MMI is not algorithmically particularly complex, the amount of the trivial code makes it large indeed. On the other hand, the division inside the DSP segment reflects well PHUZOR's synthesis architecture complexity. Rather surprising is the amount of Event Handling code, which is mere 3 % of the entire lot.

The total size of PHUZOR may be proportioned to the source code distribution of csound v4.11 [33], which has 66968 statements (i.e. it is 3.5 times bigger than PHUZOR, even when lacking the MMI). Perry Cook's Synthesis Toolkit v4.20 [63] is object-oriented, having 98 classes and 17511 statements, which makes it roughly the same size as PHUZOR.

Chapter

8 Conclusions and Further Work

In this thesis, a generic design methodology was applied to the design of a software sound synthesizer. A composite synthesis architecture comprising several elementary synthesis techniques and sound processing algorithms was proposed and used as a conceptual model. A prototype software sound synthesizer was then implemented as a VST plugin, using C++ as the programming language.

PHUZOR includes all the features that were specified for the prototype during the design phase, and it is now time to decide whether development of the final version is worth the effort. The main results of the thesis are discussed in section 8.1, and some improvements and extensions to the implemented prototype are suggested in 8.2.

8.1 Conclusions

The entire process from specification requirements, via software architectural design to the actual implementation phase was iterative. In the light of this work, ESA's software lifecycle model should best be regarded as a tool which organizes the decisions and other knowledge data gathered throughout the project into manageable chunks that can later be referred easily. The strict formalism of the produced documents was found to be too demanding to keep them up to date, should every implementation time decision be reflected back into the design model. When working with the actual release version, the amount of extra work might be considered beneficial, however.

The use case approach (i.e. snapshots of working scenarios) was applied only to the generic synthesizer model, as the functional separation worked best when designing the synthesis architecture of the dedicated device. In general, the object-oriented approach suits perfectly to the modeling of synthesizer elements, so it is a pity that the real-time demands require it to be often sacrificed for the sake of efficiency.

Considering PHUZOR, the architectural design phase of the software succeeded relatively well, because there are only few links between subsystems and classes, and most of the detail is hidden inside the boxes. This was also noticed during implementation phase.

One of the main objectives of this thesis was to find a conceptual model that fuses several elementary synthesis techniques into a single compound semi-modular synthesis architecture. Although the final judgement can be made only after more patches have been created, the audio rate modulation between particles, simple arithmetic mixing and

the possibility to exploit single synthesis method in detail appears to offer such a model. The actual synthesis architecture of PHUZOR seems flexible, but at the same time rigid enough so that user is not overwhelmed with patching possibilities. The concept of freely assignable modulators would perhaps been better if some of the most common routings were hardcoded, and only a subset of the modules was routed via modulation matrix.

DSP code is usually very tight, but the amount of MMI code was still a surprise. Using a 3rd party widget library is not enough to make significant code size reduction, as much of the code is required to handle particular implementation specific logic. It is hoped that the framework will give some assistance for future projects. The real-time nature of the implementation also makes debugging a very difficult task. It was found that ears, oscilloscope and a spectrum analyzer were the best tools available.

It shall be remained to be heard if PHUZOR has a characteristic sound of its own, or whether it can just mimic the timbres of others. It has the parameters that my personal dream machine would have, but other sound designers might face the same kind of shortcomings that led me into the development of PHUZOR in the first place. The prototype has nevertheless proved that a finalized version is realizable.

8.2 Further Work

Although the overall concept of PHUZOR is already in its final stage, in order to build a release quality software synthesizer, some polishing actions need still to be carried out. In particular, source code optimization and slight MMI related improvements would make it more usable in sequencing and live performance scenarios. A release related issue would also be the replacement of the VST wrapper component into another plugin architecture.

A versatile patch collection and a sample library should also be available. As PHUZOR's synthesis architecture seems to be quite flexible and consequently be able to broad-mindedly emulate other synthesizer models, an offline application that converts pre-programmed patches into PHUZOR format could be developed. The hierarchical patch concept demands furthermore a dedicated MMI component to be implemented, as subpatch handling at the moment is scattered around the interface.

It would be interesting to include additional synthesis techniques and modifier algorithms into the framework. The pluck particle was from the start regarded as a placeholder for a collection of more refined physical modeling algorithms, and more research is required also for the already included Bézier and XOR modulation methods. A standalone version of PHUZOR could serve as a host for other plugin effects as external modifiers or synthesizers as external particles, and to complete the circle, it could even host a copy of itself.

Chapter

9 References

- [1] C. Roads, *The Computer Music Tutorial*, MIT Press, 1996.
- [2] C. Dodge and T.A. Jerse, *Computer Music: Synthesis, Composition and Performance*, 1st Edition, Schirmer Books, New York, 1985.
- [3] J. A. Moorer, "Signal Processing Aspects of Computer Music: A Survey", *Proceedings of the IEEE*, vol. 65, No. 8, pp. 1108-1137, August 1977.
- [4] T. Tolonen, V. Välimäki and M. Karjalainen, *Evaluation of Modern Sound Synthesis Methods*, Report 48, Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology, 1998.
- [5] A. Horner and L. Ayers, "Modeling Acoustic Wind Instruments with Contiguous Group Synthesis", *J. Audio Eng. Soc.*, vol. 46, No.10, pp. 868-879, October 1998.
- [6] V. Välimäki, "Discrete-Time Synthesis of the Sawtooth Waveform With Reduced Aliasing", *IEEE Signal Processing Letters*, vol. 12, No. 3, pp. 214-217, March 2005.
- [7] E. Brandt, "Hard Sync Without Aliasing", *Proc. 2001 International Computer Music Conference*, (Havana, Cuba), ICMA, pp. 365-368, 2001.
- [8] R.A. Moog, "A Voltage-Controlled Low-Pass High-Pass Filter For Audio Signal Processing", Presented at the *AES 17th Annual Meeting*, Preprint 413, October 1965.
- [9] Hewlett-Packard, *Amplitude and Frequency Modulation*, Application Note 150-1 (HP publication number 5954-9130), 1996, Available on-line at contact.tm.agilent.com, Referenced 13.05.2005.
- [10] J. Chowning, "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation", *J. Audio Eng. Soc.*, vol. 21, No.7, pp. 526-534, September 1973.
- [11] J.-P. Palamin, P. Palamin and A. Ronveaux, "A Method of Generating and Controlling Musical Asymmetrical Spectra", *J. Audio Eng. Soc.*, vol. 36, No.9, pp. 671-685, September 1988.
- [12] Tan et al., "Real-Time Implementation of Double Frequency Modulation (DFM) Synthesis", *J. Audio Eng. Soc.*, vol. 42, No.11, pp. 918-926, November 1994.
- [13] B. Schottstaedt, "The Simulation of Natural Instrument Tones Using Frequency Modulation with a Complex modulating Wave", *Computer Music J.*, vol. 1, No.4, pp. 46-50, 1977.
- [14] D. Arfib, "Digital Synthesis of Complex Spectra by Means of Multiplication of Nonlinear Distorted Sine Waves", *J. Audio Eng. Soc.*, vol. 27, No.10, pp. 757-768, October 1979.
- [15] M. LeBrun, "Digital Waveshaping Synthesis", *J. Audio Eng. Soc.*, vol. 27, No.4, pp. 250-266, April 1979.

-
- [16] J. Beauchamp, "Brass Tone Synthesis by Spectrum Evolution Matching with Nonlinear Functions", *Computer Music J.*, vol. 3, No.2, pp. 35-43, 1979. Reprinted in [17].
 - [17] C. Roads ed., *Foundations of Computer Music*, MIT Press, 1985.
 - [18] B. Lang, "Waveform Synthesis using Bezier Curves with Control Point Modulation", *AES Convention Paper 6044*, Presented at the 116th AES Convention, 2004 May 8-11, Berlin, Germany.
 - [19] Wolfram Research, *Bézier Curve*, Mathworld, <http://mathworld.wolfram.com/BezierCurve.html>, Referenced 13.05.2005.
 - [20] K. Karplus and A. Strong, "Digital Synthesis of Plucked-String and Drum Timbres", *Computer Music J.*, vol. 7, No.2, pp. 43-55, 1983. US patent number 4,649,783.
 - [21] D. Jaffe and J. Smith, "Extensions of the Karplus-Strong Plucked String Algorithm", *Computer Music J.*, vol. 7, No.2, pp. 56-69, 1983.
 - [22] C. Sullivan, "Extending the Karplus-Strong Algorithm to Synthesize Electric Guitar Timbres with Distortion and Feedback", *Computer Music J.*, vol. 14, No.3, pp. 26-37, 1990.
 - [23] *SoundFont Technical Specification*, version 2.01, July 23, 1998. Available on-line from <http://soundblaster.com/soundfont/faqs/sfspec21.pdf>, Referenced 14.05.2005.
 - [24] *Downloadable Sounds Level 1 Specification*, version 1.1b, September 2004, Midi Manufacturers Association, <http://www.midi.org/about-midi/dls/abtdls.shtml>, Referenced 21.07.2005.
 - [25] *MPEG-4 Structured Audio homepage*, <http://sound.media.mit.edu/mpeg4>, Referenced 14.05.2005.
 - [26] *The Complete MIDI 1.0 Detailed Specification*, <http://www.midi.org/about-midi/specinfo.shtml>, Referenced 16.05.2005.
 - [27] *E-MU Desktop Instruments*, <http://www.emu.com/products/product.asp>, Referenced 15.05.2005.
 - [28] Roland Corporation, *V-Synth v2.0 Owner's Manual*, reference number 03903423, 2005.
 - [29] Yamaha Corporation, *Motif Owner's Manual*, reference number 202MWAP15.2-06E0, 2001.
 - [30] Korg Inc., *Triton Studio Parameter Guide*, 2002.
 - [31] Korg Inc., *EXB-MOSS Owner's Manual*, 2002.
 - [32] J. McCartney, "Rethinking the Computer Music Language: SuperCollider", *Computer Music J.*, vol. 26, No.4, pp. 61-68, 2002.
 - [33] *Csound Homepage*, <http://www.csounds.com/>, Referenced 29.05.2005.
 - [34] *Pure Data Portal*, <http://www.puredata.org/>, Referenced 29.05.2005.
 - [35] *Native Instruments Reaktor 5*, <http://www.native-instruments.com>, Referenced 29.05.2005.
 - [36] *KVR Audio*, <http://www.kvraudio.com>, Referenced 02.06.2005.
 - [37] *Steinberg Support for 3rd Party Developers*, <http://www.steinberg.de/Steinberg/Developers483b.html>, Referenced 15.05.2005.

-
- [38] *Cakewalk DXi SDK*, <http://www.directxfiles.com/devxchange/dxi/default.asp>, Referenced 29.05.2005.
 - [39] *Apple CoreAudio SDK*, [http:// developer.apple.com/sdk/](http://developer.apple.com/sdk/), Referenced 29.05.2005.
 - [40] *Linux Audio Developer's Simple Plugin API*, <http://www.ladspa.org/>, Referenced 29.05.2005.
 - [41] VirSyn Software Synthesizer, *TERA 2.0*, http://www.virsyn.de/en/E_Products/E_TERA/e_tera.html, Referenced 21.07.2005.
 - [42] *Software Engineering Standards*, Issue 2, February 1991 (superseded by ECSS-E-40), ESA Board for Software Standardization and Control (BSSC). Available from http://www.esa.int/SPECIALS/ESA_Publications/index.html.
 - [43] M. Fowler and K. Scott, *UML Distilled*, 2nd edition, Addison-Wesley Professional, 1999.
 - [44] *Microsoft Multimedia Standards Update*, Revision 3.0, April 15, 1994. Available online at www.tsp.ece.mcgill.ca/MMSP/Documents/AudioFormats/WAVE/Docs/RIFFNEW.pdf, Referenced 15.05.2005.
 - [45] *Microsoft Developers Network Library*, <http://msdn.microsoft.com/library/default.aspx>, Referenced 15.05.2005.
 - [46] *Wavefiles*, Sonic Spot, www.sonicspot.com/guide/wavefiles.html, Referenced 15.05.2005.
 - [47] *SDIF Homepage*, <http://www.cnmat.berkeley.edu/SDIF/>, Referenced 15.05.2005.
 - [48] J. Kleimola, "*PHUZOR Requirements Specification Document*". Unpublished.
 - [49] Yamaha Corporation, *DX7 Operating Manual*, available from <http://www.yamaha.co.jp/manual/english>, Referenced 21.07.2005.
 - [50] T. Fleischer, *MiniHost*, available from www.tobybear.de/p_minihost.html, Referenced 24.07.2005.
 - [51] Sony Corporation, *Sound Forge*, available from <http://www.sonybiz.net/proaudio>, Referenced 24.07.2005.
 - [52] M. Tippach, *ASIO4ALL*, available from <http://www.tippach.net/asio4all>, Referenced 24.07.2005.
 - [53] J. Kleimola, "*PHUZOR Architectural Design Document*". Unpublished.
 - [54] W. H. Press et al, "*Numerical Recipes in C*", Cambridge University Press, 1992, available in pdf format from <http://www.library.cornell.edu/nr/bookcpdf.html>, Referenced 04.08.2005.
 - [55] O. Matthes, *freeverb~*, MSP external port of Jezar's Freeverb, available from <http://www.akustische-kunst.org/maxmsp/index.html>, Referenced 07.08.2005.
 - [56] T. Stilson and J. Smith, "*Analyzing the Moog VCF with Considerations for Digital Implementation*", Presented at the 1996 ICMC, Hong Kong, available online from <http://ccrma-www.stanford.edu/~stilti/papers/Welcome.html>, referenced 12.08.2005.
 - [57] A. Huovilainen, "Non-linear Digital Implementation of the Moog Ladder Filter", in *Proc. 7th International Conference on Digital Audio Effects (DAFx'04)*, pp. 61-64, Naples, Italy, October 5-8, 2004, available from http://dafx04.na.infn.it/WebProc/Proc/P_061.pdf, referenced 12.08.2005.
 - [58] F.R. Moore, *cmusic homepage (CARL)*, <http://www.crca.ucsd.edu/cmusic/cmusic.html>, Referenced 12.08.2005.

-
- [59] The MathWorks, *MATLAB homepage*, www.mathworks.com, Referenced 14.08.2005.
 - [60] analoguesamples.com, *minimoogman.zip*, file *saw-1osc.wav*, available from <http://www.analoguesamples.com/asamples.asp?category=Minimoog>, Referenced 14.08.2005.
 - [61] Native Instruments, *FM7*, http://www.nativeinstruments.de/index.php?id=fm7_us, Referenced 14.08.2005.
 - [62] Campwood Software, *SourceMonitor version 2.0*, <http://www.campwoodsw.com/sm20.html>, Referenced 14.08.2005.
 - [63] P.R. Cook, *Real Sound Synthesis for Interactive Applications*, AK Peters, 2002.

Appendix

10 Appendix A -- Parameters

Following tables list synthesis parameters of PHUZOR showing each name, range, type, modulation destination indication (x), and comments.

Source Section

ARM Matrix Block			P1..P8	
active	0..1	bool		off, on
source particles		list		particles connected to this ARM input
modulation type	0..9	enum		FM, AM, RM, XOR, OR, AND, Mix, Sync, Pass
mod input level	0..100	int	x	

Particle Common			P1..P8	
active	0..1	bool		off, on
name		string		
modifier block		object		see Modifier Block
class	1..3	enum		wavetable, waveshaper, plucked
keyzone low	0..127	int		keyzone lower split point
keyzone high	0..127	int		keyzone upper split point

DCO Common			P1..P8	
waveform	0..7	enum/file		sin, tri, squ, pulse, saw, noise, samples
pitch ratio	0 .. 32.9999	float	x	relative to fundamental frequency, if 0 : fixed
pitch offset / fixed	-999.99 .. 9999.99	float		Hz, constant added after ratio scaling
fatness spread	0..3	int		number of pseudo oscillators / 2
fatness detune hi	-100..100	int	x	cents, for pseudo oscs above real oscillator
fatness detune lo	-100..100	int	x	cents, for pseudo oscs below real oscillator
fatness level	0..100	int	x	0 : real osc only, 100 : pseudo oscs only

DCO Wavetable			P1..P8	
phase	0..360	int	x	for singlecycle waveforms
pulse width	0..100	int	x	%, for pulse waveform only
loop mode	0..3	enum		for multicycle waveforms : off, fwd, alter, bwd
loop start point		int		in samples, for multicycle waveforms
loop end point		int		in samples, for multicycle waveforms
start offset		int		in samples, for multicycle waveforms

DCO Waveshaper			P1..P8	
mode	0..1	enum		Chebyshev, Bézier
harmonic coefficients		list		in Chebyshev mode
(x1,y1) and (x2,y2)		int	x	control points for Bézier

DCO Plucked			P1..P8	
attack strength	0..100	int	x	affects excitation spectrum
attack length	0..100	int	x	number of repeats of excitation wave
repluck count	0..10	int	x	number of replucks

repluck time	0..1000	int	x	msecs, time interval between replucks
pick material	0..100	int	x	0 : hardest, 100 : softest plectrum
pick position	0..100	int	x	
decay	-50..+50	int	x	-50 : shortest, 0 : standard, 50 : stretched most
string stiffness	0..100	int	x	
feedback gain	0..100	int	x	
feedback pitch		int	x	

DCA		P1..P8		
output level	0..100	int	x	
feedback level	0..100	int	x	for self modulation
pan	-50..+50	int	x	if Line Mixer channel is stereophonic

Line Mixer Section

Channel		C1..C8		
active	0..1	bool		off, on
stereo	0..1	bool		mono, stereo
modifier block		object		see Modifier Block
solo	0..1	bool		

Output		send1, send2, master mixer		
mute	0..1	bool		muted, not muted
level	0..100	int	x	
pan / balance	-50..+50	int	x	-50 : full left, 0 : center, + 50 : full right
send position	0..1	bool		pre / post fader, for sends only

Master Mixer Sect

Channel		3 inputs (mix, aux1, aux2) + 1 out (master)		
active	0..1	bool		off, on
modifier block		object		see Modifier Block
level	0..100	int	x	
balance	-50..+50	int	x	-50 : full left, 0 : center, + 50 : full right

Modifier Block

topology	0..1	enum		serial, parallel
----------	------	------	--	------------------

Modifier Common		A and B		
active	0..1	bool		off, on
dry - wet balance	0..100	int	x	0 : dry only, 100 : all wet
algorithm	0..4	enum		Filter, Delay, Spatial, Waveshaper, Modulation

DCF (Filter)				
subType		enum		Moog, MoogLadder, Biquad, ...
mode		enum		LP, HP, BP, BR
slope	6..24	enum		6, 12, 18, 24 dB, where applicable
cutoff / center freq	0..100	int	x	mapped to 0..Nyquist
resonance amount	0..100	int	x	

Delay				
time		int	x	msecs, stereo delay has separate L+R
feedback level	0..100	int	x	%, stereo delay has separate L+R
cross feedback level	0..100	int	x	for stereo delay : L+R
wet pan	0..100	int	x	for stereo delay : L+R

Reverb				
room size		int	x	%
damping	0..100	int	x	%, high-frequency absorbtion
width	0..100	int	x	%, for stereo reverb

Waveshaper				
subType		enum		poly, 2seg, dist, ...
coefficients		list		polynomial coefficients
breakpoint	0..100	int	x	% for 2-segment linear
slope1	0..1	int	x	for 2-segment linear, before breakpoint
slope2	0..1	int	x	for 2-segment linear, after breakpoint
paramA..D	0..100	int	x	for preset transfer functions

Chorus + Flanger				
rate		float	x	Hz, modulator LFO's rate
depth	0..100	int	x	%
delay time		int	x	msecs
feedback level	0..100	int	x	%

Phaser				
rate		float	x	Hz, modulator LFO's rate
depth	0..100	int	x	%
width	0..100	int	x	%
feedback level	0..100	int	x	%
stages	4..12	int		number of allpass filters

Modulation Sect

EG				
active	0..1	bool		off, on
loop start	0..64	int		0 = oneshot
loop end	0..64	int		loop end = loop start : conventional sustain
numSegments	1..64	int		
EG segment			1..numSegments	
time		float	x	in msecs
level	0..1	float	x	
slope	0..1	float		0.5 : lin, < 0.5 : exp, > 0.5 : inv exp

LFO				
active	0..1	bool		off, on
waveform	0..6	enum		sin, tri, pul, saw up, saw dn, rand, wav
rate		float	x	in Hz
delay time		float	x	msecs
fadein time		float		msecs, ramp from 0 to 100 %
freerun	0..1	bool		no, yes : restart at key down -switch
initial phase	0..360	int		degrees

RVG				
active	0..1	bool		off, on
trigger mode	0..2	enum		midi, tick, LFO
counter	1..100	int	x	
threshold	0..127	int		for midi triggers, 0 : not in use

CRM Matrix Slot		1..N	
active	0..1	bool	off, on
source class	0..4	enum	EG, LFO, RVG, Midi, Widget
source	0..N	int	unique amongst source class
dest class	0..5	enum	ARM, DCO, MFX, DCA, MOD
dest subclass		enum	depends on dest class
destination	0..N	int	unique amongst dest class + subclass pair
amount	-100..+100	int	x %, negative changes source polarity
curve		enum	lin, log, exp, ...