Jaakko Pero

Audio Programming Interfaces in Real-time Context

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo, 25.8.2014

Thesis supervisor:

Prof. Vesa Välimäki

Thesis instructor:

Tapani Pihlajamäki, M.Sc. (Tech.)

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Jaakko Pero

Title: Audio Programming Interfaces in Real-time Context

Date: 25.8.2014 Language: English Number of pages:10+74

Department of Signal Processing and Acoustics

Professorship: Acoustics and Audio Signal Processing Code: S-89

Supervisor: Prof. Vesa Välimäki

Instructor: Tapani Pihlajamäki, M.Sc. (Tech.)

In this thesis, three popular, generally available audio programming interfaces, ALSA, Core Audio, and WASAPI, were compared. A modified real-time Karplus-Strong plucked-string model application was implemented using all three APIs. In order to compare the performances, the wavetable of the plucked-string model was effectively replaced by the unknown delay of the system. In the tests, a short burst of white noise was written to the physical audio output device of the sound card, which was hardwired with a short cable to the input port of the same device. The input stream was then acquired by the application, stored on an additional buffer for further analysis, but also sent back to the output device, in order to create a loop. The noise burst in the loop acts similarly to a string instrument after its initial excitation. As the model is run in real-time, the latency of the whole system appears as the length of the wavetable. These latencies were compared.

In order to guarantee a fair comparison, the applications and corresponding operating systems were installed and run natively on the same Apple hardware, without additional virtualization layers. The runs were recorded and latencies were determined by analyzing the recordings. By compensating the known effect of buffer size and sample rate, the overhead latency characteristic of each implementation was extracted from the results. Overhead latencies were found to be within a few milliseconds. The smallest overhead latencies were measured from the ALSA implementation at 96 kHz. Overall, ALSA gave the best performance, and WASAPI was nearly as good. The largest overhead latencies were measured from the Core Audio implementation both at 44.1 kHz and 48 kHz sample rates.

Additionally, the APIs were compared in terms of major existing API design recommendations. The steepness of the learning curve of an API can be estimated by counting the number of methods the programmer is exposed to. Compared with the other two, ALSA was found to expose a significantly larger number of methods.

Keywords: Audio Processing, Digital Signal Processing, Latency

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Jaakko Pero

Työn nimi: Reaaliaikaisten audio-ohjelmointirajapintojen vertailu

Päivämäärä: 25.8.2014 Kieli: Englanti Sivumäärä:10+74

Signaalinkäsittelyn ja akustiikan laitos

Professuuri: Akustiikka ja äänenkäsittelytekniikka Koodi: S-89

Valvoja: Prof. Vesa Välimäki

Ohjaaja: DI Tapani Pihlajamäki

Tässä työssä vertailtiin kolmea yleisesti saatavilla olevaa audio-ohjelmointirajapintaa, ALSAa, Core Audiota ja WASAPIa. Suorituskyvyn vertailemiseksi toteutettiin Karplus-Strong-kielimalliin perustuva reaaliaikainen testiohjelma kaikilla kolmella eri ohjelmointirajapinnalla. Mallin aaltotaulukko korvattiin järjestelmän tuntemattomalla viiveellä. Testeissä lyhyt valkoisen kohinan purske lähetettiin äänikortin ulostuloon, joka oli kytketty saman laitteen sisäänmenoon lyhyellä kaapelilla. Testiohjelma luki signaalia laitteen sisäänmenosta, tallensi sen analyysiä varten, mutta lisäksi lähetti sen alipäästösuodattimen läpi takaisin ulostuloon, muodostaen silmukan. Kohinapurske käyttäytyy tällaisessa silmukassa kuin kielisoittimen kieli, kun se on ensin alkutilastaan poikkeutettu. Kun testiohjelmaa suoritettiin reaaliaikaisesti, järjestelmän viiveet eli latenssit näkyivät kielimallissa aaltotaulukon pituuksina, joita vertailtiin.
Testiohjelmia ajettiin samalla Apple-laitteistolla, ajot tallennettiin ja latenssit määritettiin tallenteista. Virtualisointiohjelmia ei käytetty, vaan käyttöjärjestelmiä ohjelmistoineen ajettiin laitteessa sellaisenaan. Mittauspisteet valittiin siten, että samoja voitiin käyttää kaikkien toteutusten mittauksissa. Toteutusten keskinäistä vertailua varten puskurikoon ja näytteenottotaajuuden vaikutukset vähennettiin tuloksista, jolloin jäljelle jääneiden latenssien havaittiin olevan muutaman millisekunnin sisällä toisistaan. Pienimmät latenssit mitattiin ALSAa käyttäneellä toteutuksella 96 kHz:n näytteenottotaajuudella. ALSAlla saavutettiin yleisesti paras suorituskyky, ja WASAPIllakin lähes yhtä hyvä. Suurimmat latenssiarvot mitattiin Core Audio -rajapintaa käyttäneellä toteutuksella.
Lisäksi rajapintoja vertailtiin suhteessa yleisiin suunnitteluperiaatteisiin. Vertailussa huomioitiin metodien määrä ja arvioitiin käyttöönoton helppoutta ja dokumentaation saatavuutta. Suuri opeteltavien metodien määrä hidastaa rajapinnan käyttöönottoa. Rajapintojen metodit laskettiin, jolloin ALSAssa havaittiin olevan muihin rajapintoihin verrattuna huomattavasti suurempi määrä ohjelmoijalle näkyviä metodeja.

Avainsanat: digitaalinen signaalinkäsittely, latenssi, äänenkäsittely

Acknowledgments

I wish to thank Jussi Pekonen and Tapani Pihlajamäki for extraordinary support. Many thanks go to my thesis supervisor, professor Vesa Välimäki. Many thanks more go to my Maija for everything, support and proofreading. Special thanks go to all my fellow students and co-workers at Aalto University. Additionally, I wish to thank Esko Järnfors for proofreading.

Otaniemi, August 25, 2014

Jaakko Pero

Contents

Symbols

e	Euler's number
E	elastic modulus
f_S	sampling frequency, sample rate
M	molecular mass in kilograms per mole
R	molar gas constant
s	unbiased sample variance
T	tension
T_S	sampling period
v	speed in meters per second

γ	adiabatic index
μ	mass of the solid per unit of length
ρ	density of the solid

Abbreviations

ADC	Analog-to-Digital Conversion
API	Application Programming Interface
APU	Audio Processing Unit
ASIO	Audio Stream Input/Output
CD-DA	Compact Disc Digital Audio
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DAC	Digital-to-Analog Conversion
DFT	Discrete Fourier Transform
DRM	Digital Rights Management
DTS	Digital Theater Systems
FFT	Fast Fourier Transform
GCC	GNU project C and C++ compiler
GNU	GNU's Not Unix!
HAL	Hardware Abstraction Layer
HDMI	High-Definition Multimedia Interface
IDE	Integrated Development Environment
I/O	Input/Output
JACK	JACK Audio Connection Kit
LTI	Linear Time-Invariant
MIDI	Musical Instrument Digital Interface
MSDN	Microsoft Developer Network
OS	Operating System
PC	Personal Computer
PCM	Pulse-Code Modulation
RISC	Reduced Instruction Set Computer
SDL	Simple DirectMedia Layer
SNR	Signal-to-Noise Ratio
SOC	System On a Chip
STFT	Short-Time Fourier Transform
S/PDIF	Sony/Philips Digital Interconnect Format
VST	Virtual Studio Technology
WASAPI	Windows Audio Session API
x86	Popular CISC architecture of Intel Corporation

List of figures

List of tables

Chapter 1

Introduction

Real-time audio programming has been a notoriously difficult task. Until the 1990s, the application programming interfaces of different audio cards were largely incompatible. As digital audio was not the primary use of a desktop workstation, many operating systems only supported a very limited PC-speaker sound, or no audio features at all. Audio programmers had to target their applications at a specific brand of sound card or even at individual device models.

As an opposite to this, modern operating systems tend to mask the exact hardware details and it has become unnecessary or even impossible to write device specific code. Complex operations can be executed in real-time reliably, even if the source code is written in a language with a high level of abstraction. Typically, the operating system provides the programmer with at least one low-level audio interface that is used as a foundation for higher levels of abstraction.

Many successful general audio programming interfaces that provide similar functionality, coexist. Additionally, most of them are cross-platform development kits that reduce the amount of resources needed to port an application to another platform. At their best, they give the audio programmer an opportunity to focus on productive audio processing as hardware peculiarities are left below the abstraction layers.

For two decades, personal computers have been used as relatively inexpensive, general-purpose audio production and playback equipment. The success of the Intel x86 as a target platform was and is due to its (backwards) compatibility and versatility. An apparent typewriter could be easily upgraded into a professional audio workstation simply by installing an audio interface card. Undoubtedly, the ability to run the latest 3D shooter game had an effect, too. In the field of music industry, virtual instrument technologies like Steinberg VST emerged in the late 20th century and marked an end to the dominance of specialized professional audio production equipment. (Osorio-Goenaga, 2005)

From another point of view, the advances in global standards compliance, electron-

ics and mass production have blurred the line between professional and consumer devices. Compatibility has become the norm and users expect to be able to, for example, connect their mobile gadgets directly to their studio equipment. Entry-level products from all professional audio brands and manufacturers are widely used at home by hobbyists. Internet has enabled musicians all over the world to exchange and distribute their audio tracks in a blink of an eye without having to worry about their tracks degrading from studio quality. (Rumsey, 2011)

The structure of this thesis is as follows. First, an introduction to audio processing along with some properties of sound and hearing is given in Chapter 2. In Chapter 3, various analog-to-digital and digital-to-analog converter implementations are introduced. In Chapter 4, an introduction to audio programming is given and the relation of source code and processor instructions is discussed. The aim of Chapter 5 is to study the properties that are to be considered when a programming framework is about to be selected for a project. In Chapter 6, a schematic description of a generalized real-time audio measurement application is given. A test application for latency measurements is presented in Chapter 7. In Chapter 8, the practical aspects of latency measurements are discussed, and the details of the measurement platform are examined. Additionally, the results of the latency measurements are given, along with some statistical analysis and discussion. Chapter 9 contains the conclusion and propositions for future work.

Chapter 2

Digital processing of sound

In this chapter, some properties of sound, sound waves, and hearing are studied briefly. A brief history and some fundamental properties of analog-to-digital conversion and digital audio processing are also discussed.

2.1 Some properties of the sound wave

Sound waves are longitudinal disturbances, or pressure variations, that carry information and energy in a medium. Molecules of the media swing back and forth perpendicularly to the direction of the sound wave. Typical sound sources have some directivity, but essentially they create spherical waves that expand in every direction. After traveling a sufficient distance, the wave can be approximated as a plane wave.

Waves have a number of important properties. By the principle of linear superposition, multiple waves can pass through each other and retain their properties. At the point they meet in, they interfere linearly, either additively or subtractively, based on their phase differences.

Sound waves can propagate in solid, liquid, or gas, but not in a vacuum. The speed of a sound wave v in ideal gas is

$$v = \sqrt{\frac{\gamma R \mathrm{T}}{M}}, \tag{2.1}$$

where γ is the adiabatic index, R the molar gas constant, T the absolute temperature of the gas, and M the molecular mass of the gas (Rossing et al., 2002). For air in room temperature, v is approximately 340 m/s. It should be noted that Eq. 2.1 is independent of sound frequency. For example, speech can travel long distances over air and still be sufficiently interpreted.

Sound waves can travel in solids as well, typically at higher speeds than in gases. Longitudal waves travel at speed

$$v = \sqrt{\frac{E}{\rho}}, \tag{2.2}$$

where E is the elastic modulus and ρ the density of the solid.

In solids, sound can propagate also as transverse waves, which travel at speed

$$v = \sqrt{\frac{T}{\mu}}, \tag{2.3}$$

where T is the tension, and μ mass of the solid per unit of length.

As the waves travel in media, they frequently encounter boundaries. A portion of the wave energy undergoes transmission across the boundary, whereas some of it reflects, and some is absorbed by the medium (the energy turns into heat). If a sound wave goes around a barrier or through an opening, diffraction occurs and the wave changes its direction. Refraction occurs when the speed of a wave changes as a result of changes in the parameters of the medium it passes. (Rossing et al., 2002)

In addition to the ability of sound waves to travel across the boundaries between solids and gasses, the boundaries form mechano-acoustic systems where sound energy can transform from a form to another. As a result, a vibrating cone of the loudspeaker driven by an artificial electric signal can emit sound to the surrounding environment. (Kagawa et al., 2004)

2.2 Hearing

The ear detects pressure changes. The human ear functions on a limited frequency spectrum, in the range of 20 to 20000 Hz on a young healthy person. The ability to hear higher frequencies degrades as we age. This degradation is also frequency dependent. Although the hearing abilities of elderly people at frequencies up to 400 – 500 Hz may stay about the same, the dynamic range is greatly reduced at higher frequencies. The minimum volume required to hear the sound and the volume loud enough to cause pain approach each other, as we get older. (Herold, 1988)

The study of basic physiology of the hearing mechanisms by Evans (1993) shows that the cochlea – the auditory portion of the inner ear – has three major functions that

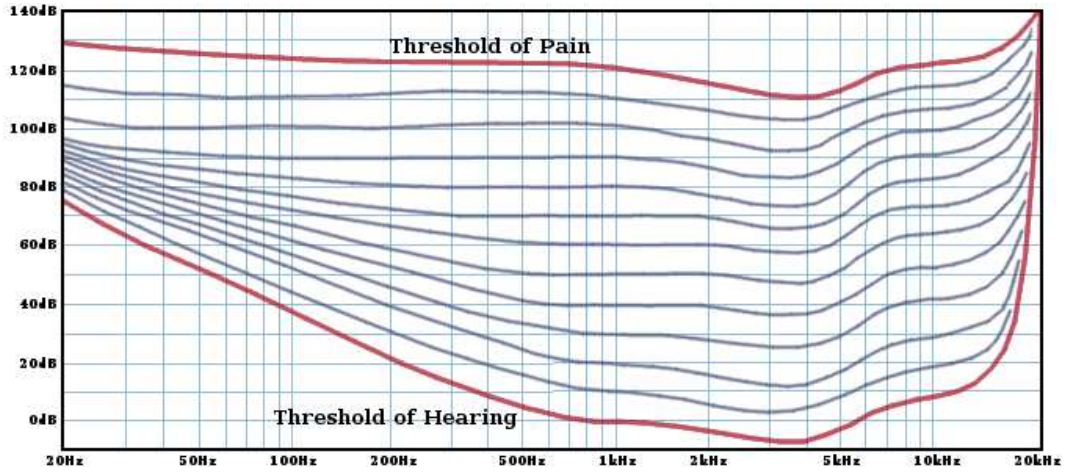

Figure 2.1: Approximate equal-loudness curves derived by Montgomery (2013) from Fletcher and Munson (1933) and modern sources for frequencies above 16 kHz.

can be simulated in hardware and software. Accordingly, many properties of hearing in general can be determined by observing cochlear nerve fibres. The ear behaves like an array of tuned bandpass filters, which enable us (mammals) to divide sounds into their individual frequency components. Furthermore, the ear can be considered as an instrument analogous to a microphone. Outer and inner ear structures filter the sound depending on, for example, the relative direction of the sound. Filtered sound enters the cochlea as fluctuations in fluid pressure, thus moving the rows of cochlear hair cells, which generate electric signals to activate the synapses. The stimuli are then conducted to the auditory brain via cochlear nerve fibres. Cochlear nerves have different, partially overlapping tunings, which span the range of hearing.

The minimum and the maximum human detectable pressure change, or the signal amplitude, is limited between approximately 0 dB, the hearing threshold, and 130 dB, the pain threshold. Often in literature, these limits are given as four numeric values (20 – 20000 Hz, 0 – 130 dB) although the phenomenon is strongly frequency dependent. As proven by Fletcher and Munson (1933), a more specific and accurate way to present the limitations of human hearing is the equal-loudness curve system, illustrated in Fig. 2.1.

The frequency dependent perception of loudness makes controlling the loudness of a sound reproduction system quite complicated. Essentially, the sensitivity of the ear falls off more in the low frequencies than in midrange. This means that if a person perceives the loudness of a playback system to be too low, it cannot be compensated by applying a scalar gain increase, as it would break the balance between (the perceived) low and high frequencies. (Holman and Kapman, 1978)

In digital recordings, in order to cover the theoretical dynamic range of human hearing, which peaks at around 1 kHz frequency, a bit depth of 20 bits per sample is needed. The term *studio quality* is often used in literature. In digital context, it

refers to audio content that has at least 20 bits of depth per channel per frame and a sample rate of 48 kHz. A frame is a term used in place of a sample when a digital audio stream consists of multiple channels. All samples in a frame are to be played simultaneously on individual channels. Accordingly, the value of the frame rate in a multichannel stream is typically the same as the sample rates of individual channels. (Grewin, 1989)

Finally, the brain interprets these signals as an auditory sensation. Again, the word *sound* is used to describe both the sensation, and the disturbance that might cause it (Rossing et al., 2002). The exact details of this complex and nonlinear process are left out as they are beyond the scope of this thesis.

2.3 Brief history of and motivation to analog-to-digital conversion

In writing this section, the author was inspired by Kester and Analog Devices Inc. (2005).

The development of digital technology is closely tied to improvements in the field of the controlled amplification of electric signals. Whereas analog filters can be built entirely from passive components that do not need any external energy, such luxury is generally not available in the digital world.

The fundamental problem of telecommunications is the transmission of a signal to a receiver far away. Amplification, in the form of a regenerative repeater, enabled the full potential of pulse-code modulation (PCM) techniques. PCM is a lossless encoding technique widely used in audio applications. It was invented by Reeves (1942). In 1956, PCM was mainly used to increase the number of simultaneous phone calls on existing copper cable pairs in metropolitan areas (of the U.S.). As predicted by Deloraine and Reeves (1965), PCM became the backbone of modern telecommunications.

In all manufacturing, the individual units slightly deviate from specifications. Digital circuits are by design tolerant to parameter deviation, whereas in analog design the changes in resistor and capacitance values have an immediate effect on the performance of the unit, unless properly compensated by a more complex design. In addition, the operation of digital circuits is tolerant of the exact values of the signals. Therefore, the digital circuitry can be mass manufactured and many circuits can be easily integrated on the same chip. A digital processor can process multiple signals simultaneously via time-sharing as, by design, there is no crosstalk problem between multiple inputs and outputs. In the digital domain, the fast Fourier transform can be used to extract the digital frequency domain information for analysis or to convolve two signals cost-efficiently. Many operations are easier to implement digitally. In

summary, very complex filters with guaranteed frequency responses are easily implemented digitally.

Advantages over analog processing are, for example, that arbitrary accuracy can be achieved simply by increasing the word length, whereas the accuracy of an analog circuit is dependent on the individual components and thermal noise. The dynamic range of digital processing can be increased by choosing floating-point arithmetic. Unlike analog filters, digital filters can be cascaded without loading problems. In order to save bandwidth, various lossless as well as lossy compression techniques have been developed. Those have enabled us to transmit vast quantities of TV and radio channels by wire or wirelessly, as a broadcast, but also on demand. (Baldini, 1997)

2.4 About digital signal processing

Life is full of signals. A signal is any function of two or more independent variables that carry information. Music, speech, or any other sound is a pressure signal that varies as a function of time at a point in space. Signal processing extracts the useful information carried by a signal, independent of its original domain of existence, and represents it mathematically, often in some transformed domain.

According to Mitra (2006), the digital signal processing (DSP) workflow has several distinct steps. The first step is the sampling and holding the analog signal, illustrated in Fig. 2.2(a), over the sampling period. The resulting staircase waveform is illustrated in Fig. 2.2(b).

In the second step, the staircase waveform can then be fed to the analog-to-digital converter (ADC), which converts the waveform into a binary data stream as in Fig. 2.2(c). If the processor is next to the converter, corresponding bit switches could be connected directly. At longer distances, perpendicular wires are prone to crosstalk errors, as shown by Ott (2009). If the data is to be sent further, it is therefore encoded to a pulse-code modulation stream, PCM.

An adaptation from the original patent illustration in Fig. 2.3 provides a historic and practical example of an ADC. The device, invented by Reeves (1942), operated at a sample rate of 6 kHz and on the bit depth of only 5 bits per sample. It uses a sampling pulse to take a sample of the analog signal (VOICE INPUT), which sets a set-reset flip-flop circuit. Simultaneously, a controlled ramp voltage is started. The ramp voltage is compared with the input, and when they are equal, a pulse is generated that resets the flip-flop. The output of the flip-flop circuit is a pulse of which width is proportional to the input at the sampling instant. This output (PULSE WIDTH MODULATOR) controls a gated oscillator, and the number of pulses out of it (AND) represents the quantized value of the analog signal, which can be converted to a binary word by a counter. A master clock of 600 kHz is used, and a 100:1 divider thus

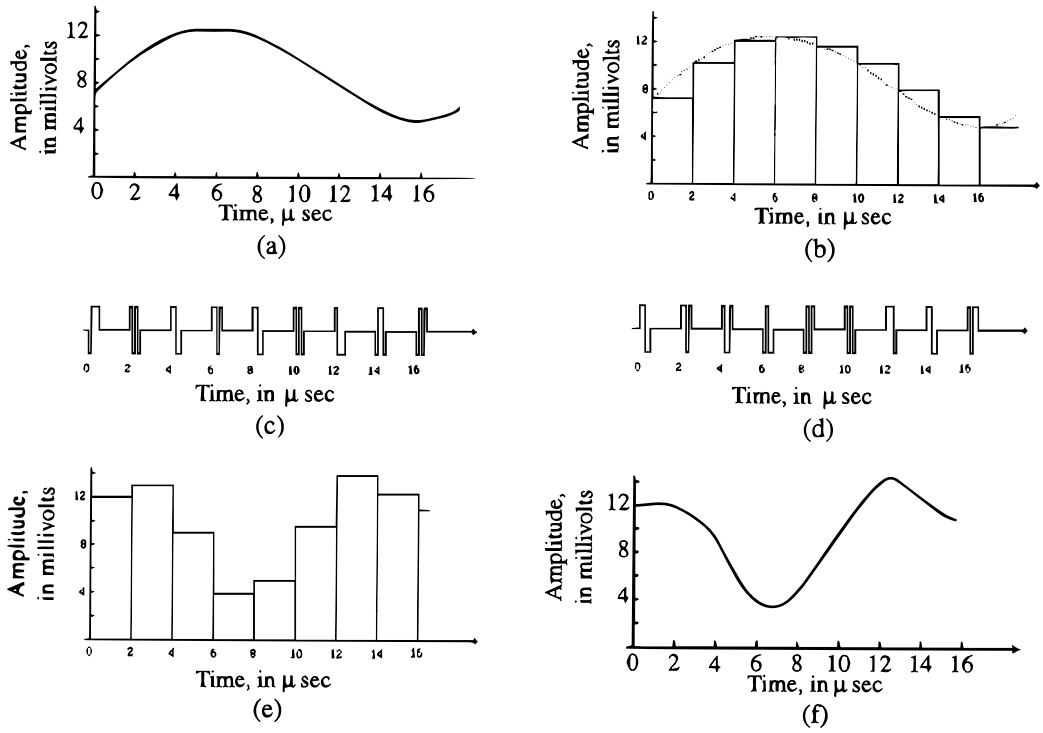

Figure 2.2: Typical waveforms at various stages of a DSP: (a) Analog input. (b) Sample-and-hold circuit output. (c) ADC output. (d) Digital processor output. (e) DAC output. (f) Analog output. Adopted from Mitra (2006).

generates the 6 kHz sampling pulses.

At this point, as shown in the middle of Fig. 2.4, the actual processing of the converted data stream can proceed in the digital domain. Operations that can be done at this point depend on the processing power available in the DSP processor, amount of time available, and the imagination of the designer.

In the third step, after the processing is done, the signal (eventually) has to be converted back to the analog domain in order for it to be heard or seen. This conversion is performed by a digital-to-analog converter (DAC). This transformation is illustrated in Fig. 2.2(d–f). First, the digital word is converted into a staircase waveform as illustrated in Fig. 2.5(e). According to Mitra (2006), the desired analog signal can be recovered from the staircase waveform via lowpass filtering. DACs are further discussed in Section 3.5.

The obvious advantage of analog processing is that an analog signal can be processed without conversions, provided an analog filter sufficient for the task ahead can be implemented. Some analog circuits can be designed from passive components so that they do not need external power, whereas digital circuits are generally active components. Active components, though, can be made very energy-efficient.

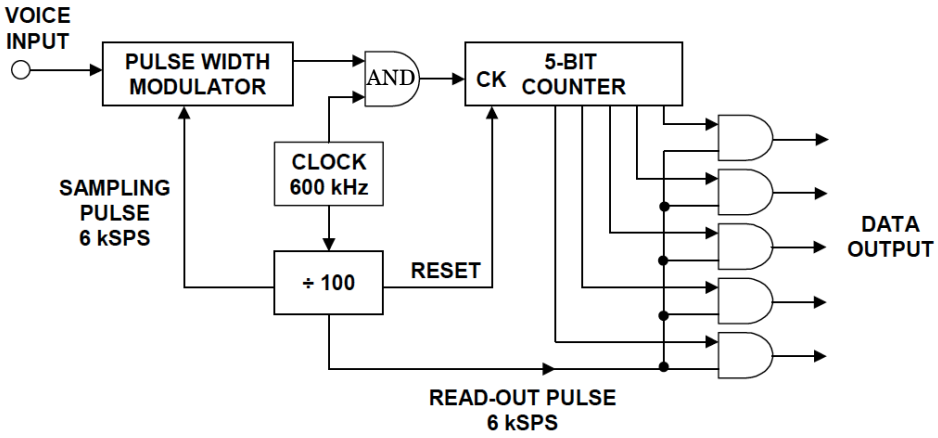

Figure 2.3: Adaptation by Kester and Analog Devices Inc. (2005) of the 5-bit counting ADC of the Reeves (1942) patent. The output of PULSE WIDTH MODULATOR is a pulse of which width is proportional to the analog input (VOICE INPUT). This output controls a gated oscillator, so that the number of pulses out of AND, which is converted to a binary word by a counter, is proportional to the input at the sampling instant.

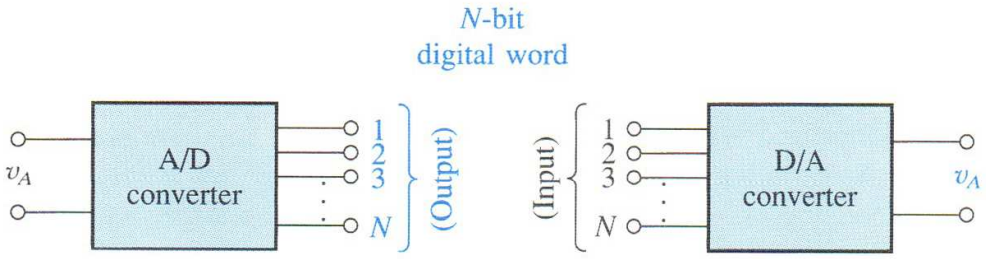

Figure 2.4: The ADC and DAC as circuit blocks. v_A represents the analog input and output signals. Adopted from Sedra and Smith (2004).

2.5 Real-time audio processing

The ability of processing signals in *real-time* is often a desired property of a DSP system. Definition of real-time in the field of audio engineering is subjective and case-specific. Physical changes have a limited speed, the highest possible speed being the speed of light in a vacuum. All digital device inputs must have a delay of at least one sample. The delay length is inversely proportional to the operating frequency. In audio devices, the operating frequencies are fairly low.

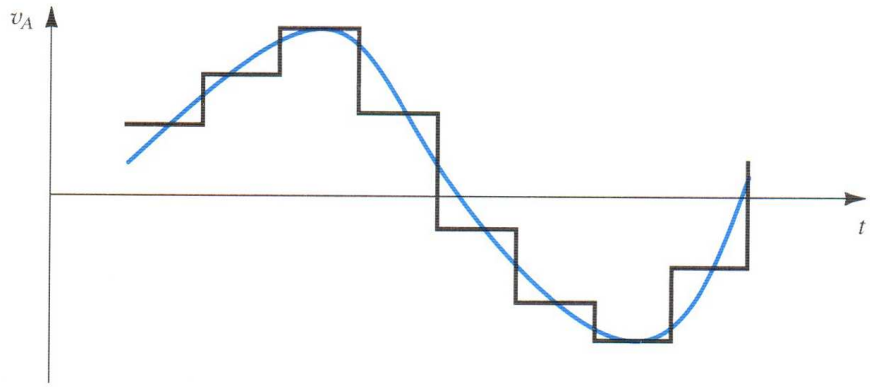

Figure 2.5: Staircase waveform v_A from a sample-and-hold circuit, as a function of time t. The corresponding analog waveform is also shown. Time delay introduced by the filter is not shown. Adopted from Sedra and Smith (2004).

According to Rossing et al. (2002), the word real-time in a field of computer music can be defined as the ability of the digital counterpart of an analog instrument to respond fast enough not to irritate the musician playing it. According to Lester and Boley (2007), the amount of latency that can be present in a signal path before musicians will actually perceive it, varies from 1 ms (in-ear monitors) to 6.5 ms (wedge monitor setup), strongly depending on the instrument. For example, keyboardists considered a 40 ms latency (an equivalent buffer size at 44100 Hz is 1764 samples) on a wedge monitoring setup still as *fair* conditions.

2.6 Linearity and time-invariance of systems

Even if not often explicitly mentioned, various mathematical features of DSP are valid only if the systems are LTI, linear and time-invariant. Definition for linearity as usually given in literature is the following: considering input functions $x_1(t)$ and $x_2(t)$, and their corresponding outputs $y_1(t)$ and $y_2(t)$ given as

$$f(x_1(t)) = y_1(t), \tag{2.4}$$

and,

$$f(x_2(t)) = y_2(t), \tag{2.5}$$

the system function f is linear if the linear combination of the inputs,

$$x(t) = \alpha x_1(t) + \beta x_2(t), \tag{2.6}$$

has a corresponding linear combination of the outputs,

$$f(x(t)) = \alpha y_1(t) + \beta y_2(t), \tag{2.7}$$

for any constants α and β (Mitra, 2006).

The linearity property enables calculations for complex input sequences that can be broken down into a weighted combination of simpler ones, and compiled back after applying the function on each sequence individually.

If an input $x_1(t)$ fed into the system f results in output,

$$f(x_1)(t) = y_1(t), \tag{2.8}$$

and the input,

$$x(t) = x_1(t - \gamma), \tag{2.9}$$

has the output,

$$f(x)(t) = f(x_1)(t - \gamma) = y_1(t - \gamma), \tag{2.10}$$

with an arbitrary sequence and constant γ, then the system f is *time-invariant*.

In layman's terms, the system, function f, is expected not to change during the time it is being observed. For example, a resistor is only time-invariant if the temperature is kept constant. If the temperature changes freely, so does the value of the resistor, and no assumption about the system impedance can be made. Digital circuits are built to be tolerant of the system temperature, but they too have safe operating temperatures that are not to be exceeded.

2.7 Mathematics of sampling

Our ability to map analog phenomena to digital domain is quite limited. Infinitely many different analog signals, when sampled at the same sample rate, map to one digital signal, of which the original analog signals cannot be restored. (Mitra, 2006)

One possible interpretation of a sampling operation is the multiplication of a continuous signal,

$$g[n] = g_a(nT_\mathrm{S}), \quad n = -\infty < n < +\infty, \tag{2.11}$$

where T_S is the sampling period, by a periodic train of ideal impulses:

$$p(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT_\mathrm{S}) \tag{2.12}$$

Considering the Dirichlet conditions (Andrews and Phillips, 2003) satisfied, so that the signal has finite discontinuities and finite number of maxima and minima in any finite interval, and that the signal is absolutely integrable, then the continuous-time Fourier transform of g_a of Eq. 2.11 yields the following frequency domain representation:

$$G_a(j\Omega) = \int_{-\infty}^{\infty} g_a(t) e^{-j\Omega t} dt \tag{2.13}$$

Correspondingly, the discrete-time Fourier transform of $g[n]$ of Eq. 2.11 gives the following frequency domain representation:

$$G(e^{j\omega}) = \sum_{n=-\infty}^{\infty} g_{[n]} e^{-j\omega n} \tag{2.14}$$

The multiplication result of the continuous time signal $g_a(t)$ by the impulse train $p(t)$ is then a continuous-time signal where uniformly spaced impulses at $t = nT$ are weighted by the sampled values $g_a(nT)$, as shown in Eq. 2.15 below.

$$g_p(t) = g_a(t)p(t) = \sum_{n=-\infty}^{\infty} g_a(nT_{\mathrm{S}})\delta(t - nT_{\mathrm{S}}) \tag{2.15}$$

The continuous time Fourier transforms for Eq. 2.15 are

$$G_p(j\Omega) = \sum_{n=-\infty}^{\infty} g_a(nT_{\mathrm{S}})e^{-j\Omega nT_{\mathrm{S}}} = \frac{1}{T_{\mathrm{S}}} \sum_{k=-\infty}^{\infty} G_a\left(j\left(\Omega + k\Omega_T\right)\right), \tag{2.16}$$

where the angular sampling frequency $\Omega_T = 2\pi/T$ and $G_p(j\Omega)$ is a sum of shifted and scaled copies of $G_a(j\Omega)$. The folding frequency (also Nyquist frequency) is then denoted $\Omega_m = \Omega_T/2$. It follows that the $g_a(t)$ can be recovered exactly from the $g_a(nT_{\mathrm{S}})$ by lowpass filtering the impulse train with an ideal filter with a gain T_{S} and cutoff frequency Ω_c greater than Ω_m and less than $\Omega_T - \Omega_m$. (Mitra, 2006)

Chapter 3

Analog-to-digital and digital-to-analog converters

In this chapter, a variety of analog-to-digital converter implementations are examined. Digital-to-analog conversion is also discussed briefly.

3.1 Into the digital

In order to transform a sound wave into a digital form, it is first detected and converted into an electrical signal. The acoustic-to-electric transducer device that does this is called a microphone. The circuitry is usually capacitively coupled, thus it does not convert the exact sound pressure level to an electrical signal. Instead, it tracks the changes, or the differential of the signal. The electrical alternating signal, consisting of current and voltage, is then converted to a digital form by the analog-to-digital converter, or ADC. Although the discussion is mostly audio-centric, the signals involved do not have to ever exist as a physical sound wave. The measurements do not necessarily involve any microphones or loudspeakers. The beauty of digital signal processing is that an arbitrary signal can be treated as if it was a naturally occurring sound.

In the conversion process, some of the information available is dropped. A single sound wave in a fluid medium is a complex phenomenon that has many measurable attributes including, but not limited to, frequency, intensity, velocity, and direction. All this is dropped and only the information about the sound pressure at one point in space, at one point in time, is left. In ADC or analog-to-digital conversion, the one-dimensional electrical signal that has a theoretically infinite resolution, is sampled at regular intervals.

Illustration of the key component of sampling process, the sample-and-hold circuit, is provided in Fig. 3.1. Analog voltage source v_I is sampled during the sampling period by the switch S being closed, as the voltage over capacitor C reaches v_I. During the

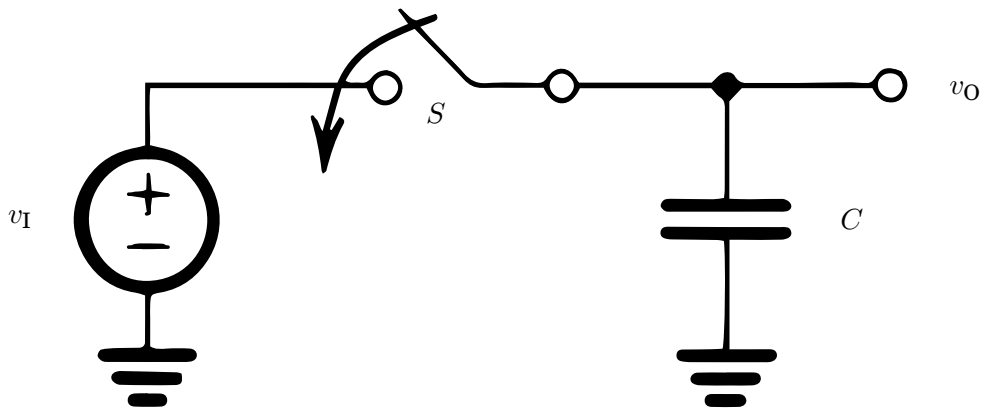

Figure 3.1: A simplified sample-and-hold circuit. During the hold period when the switch S is open, the voltage v_0 held on the capacitor represents the sample value, to be fed to the actual analog-to-digital converter circuitry. Adapted from Sedra and Smith (2004).

hold period, switch S is open. As there is no current through the capacitor, its voltage v_O can be read and stored as the *sample* of v_I.

A visualization of signals involved in a sampling process is given in Fig. 3.2. The analog input signal v_I is sampled by the sampling control signal v_S every clock period T for a smaller period τ and then held for the rest of the time $T - \tau$, thus resulting in a staircase signal v_O suitable to be sent to the actual analog-to-digital converter circuitry. In this phase, the analog staircase signal is converted to a binary word. Each bit in the binary word is controlled by a comparator using exponentially weighted reference voltages. The conversion is synchronized to the sample rate so that each digital word sent to the digital output corresponds to the value of the original input signal during one clock period.

The signal level in each sample is furthermore quantized to the nearest representable numeric value so that it can be stored as a series of binary digits, or bits. At this point, a decision is made, whether to use uniform quantization (where the quantizing intervals are equally sized) or non-uniform quantization. In the latter, smaller quantizing intervals are allocated to smaller signal values and large quantization intervals to large signal values. If the analog input signal is properly bandlimited, sampling in time is theoretically lossless, and the original signal could be reconstructed exactly from its samples. In practice, clock jitter destroys this property as well, as shown by Putzeys and de Saint Moulin (2004).

Amplitude quantization, however, is inherently lossy, being a mapping of a range of

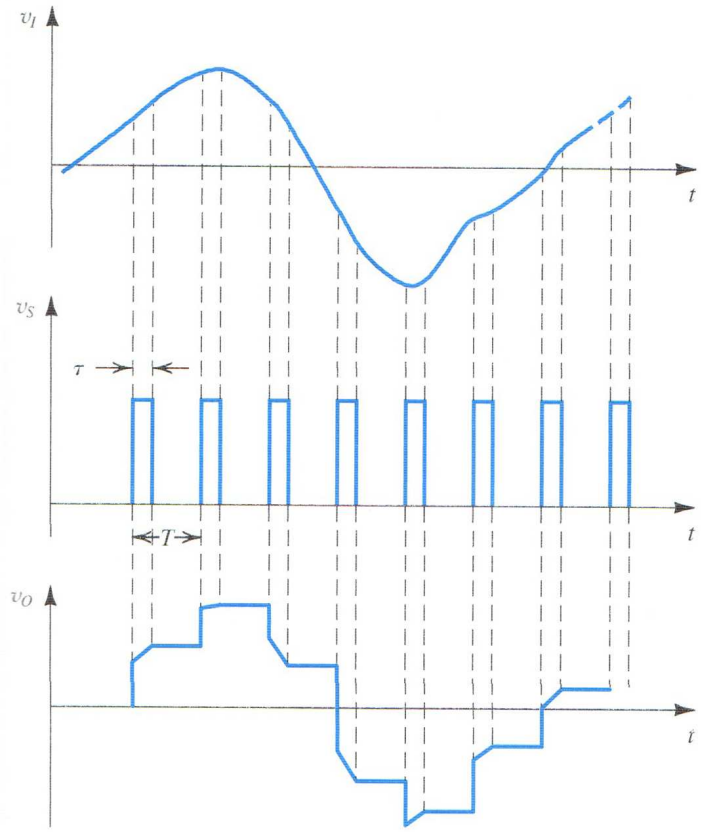

Figure 3.2: Sampling of an analog signal. Input signal v_I is sampled by the sampling control signal v_S every clock period T for a smaller period τ and then held for the rest of the time $T - \tau$, thus resulting in a staircase signal v_O suitable to be sent to the ADC circuitry. Adopted from Sedra and Smith (2004).

actual input signal values to a single output value. The unwanted correlation between quantization noise and input signal is called granulation noise. Even if the total distortion energy of a quantized sinusoidal signal is independent of the input signal amplitude, the level of individual harmonics varies significantly with changes in the input signal, contributing to a metallic timbre as the amplitude of the input signal decays. As an example of the characteristics of a quantization noise, the input-output relationship of a normalized rounding quantizer in Fig. 3.3 and the corresponding quantization error function in Fig. 3.4 is provided. (Maher, 1992)

Maher (1992) furthermore states that the quantization noise is actually odd harmonic distortion for any amplitude of an input sinusoid. Maher concludes that quantization noise is deterministic and if the input signal and quantizer characteristic are known, the error introduced in the quantizer is also known. In audio processing, this is significant, because decaying musical signals often become sinusoidal as their amplitude decreases, resulting in signal-correlated noise components at discrete frequencies.

In the digital domain, the audio samples have to have a special representation. As

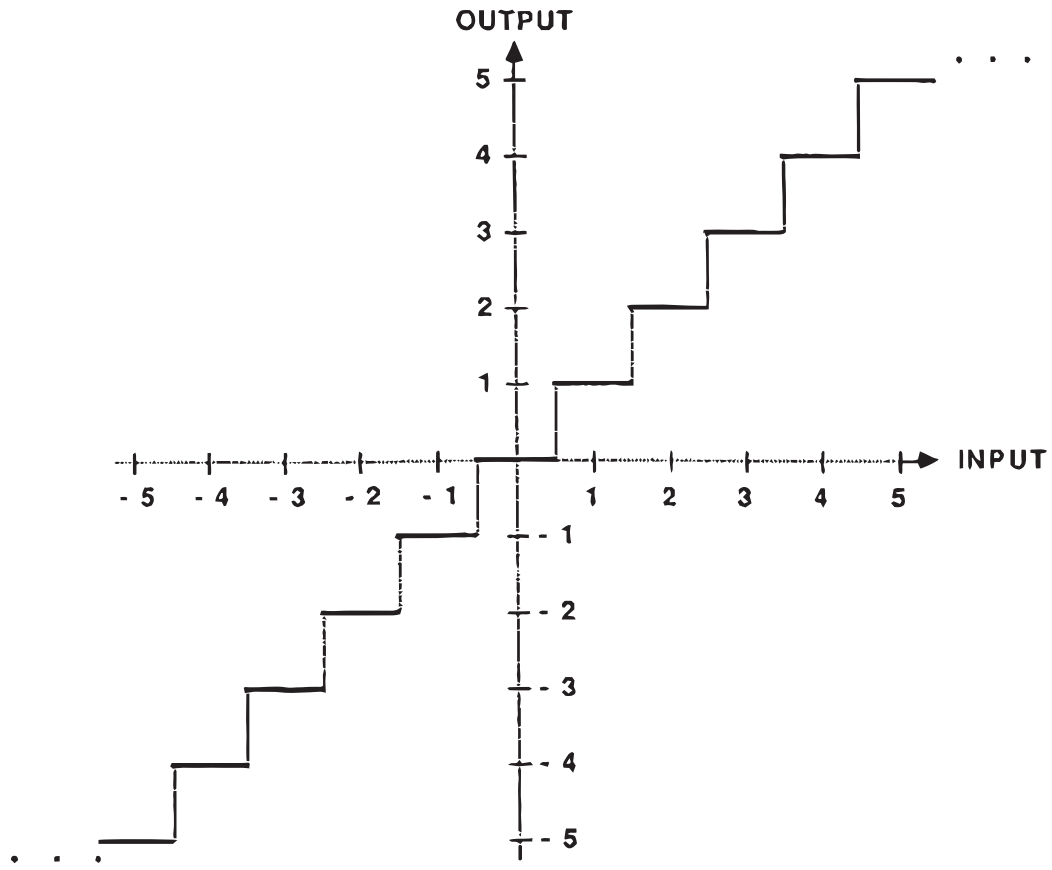

Figure 3.3: Input–output relationship of a bipolar uniform quantizer. The quantizer performs the rounding to the nearest integer operation. Adopted from Maher (1992).

the space reserved for each sample is limited, they have to have a finite amplitude resolution and the samples must represent discrete points in time. By convention, mainly because the mathematics and hardware implementations become much easier, the samples are spaced evenly in time, precisely at sampling period $1/f_{\mathrm{S}}$. There is no theoretical necessity for this, though.

ADCs generally perform the conversion in two stages. They discretize the signal in time, and then they quantize the samples in voltage. It is important that the processes are carried out in this order. Errors will be caused unless the quantizing in time is carried out first. (Story, 2004)

In the conversion process, the uncertainties and measurement errors of the analog world are in effect ignored. While we do not know the precise pressure value at a given time, we treat the resulting quantized digital sample as if we knew exactly what its value is. Furthermore, the signal values between two sampled points in time are ignored.

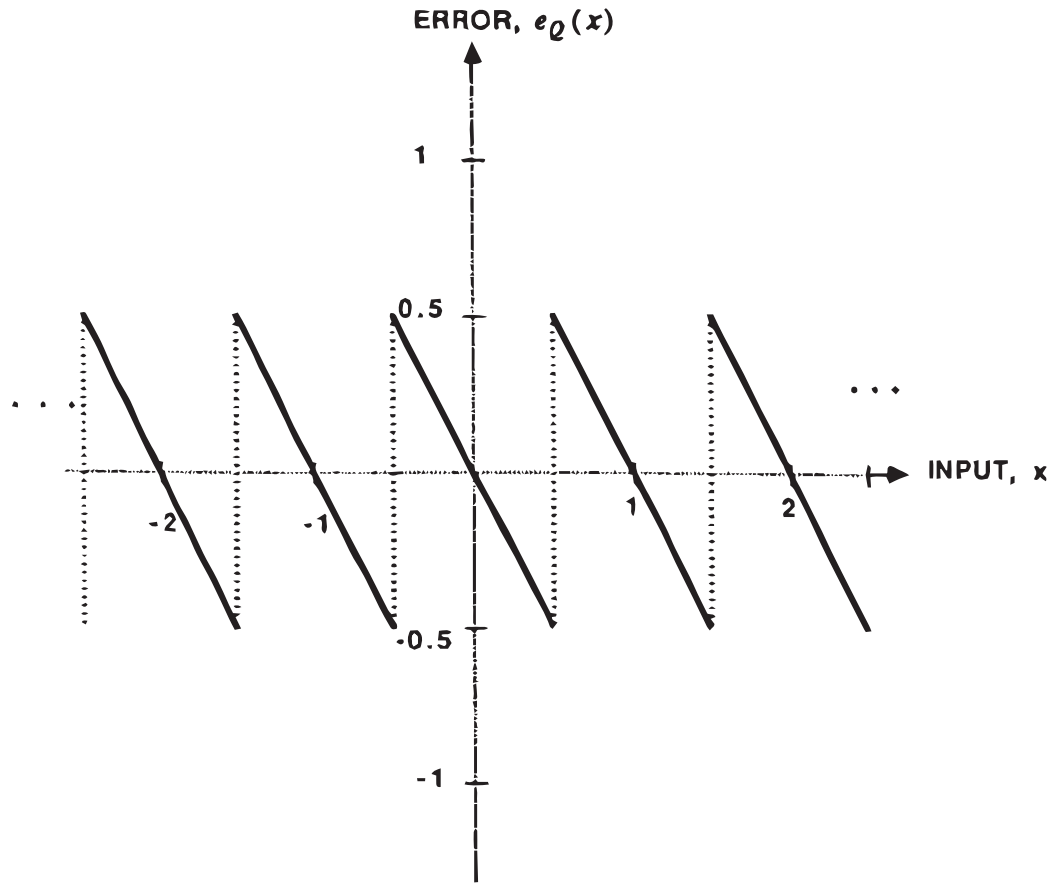

Figure 3.4: Quantization error of the bipolar uniform quantizer that has the input–output relationship illustrated in Fig. 3.3. The sawtooth function of the error signal e_Q results from the difference between the analog input signal and the quantized output signal. Adopted from Maher (1992).

Another interpretation for the phenomenon is that in the digitization process an unknown error signal is added to the perfect analog input. There are, in fact, two separate error signals. One is the discretization error, which arises from representing a function of a continuous signal on a lattice. The other is the quantization error, arising from the fact that the amplitude value of the signal cannot be stored exactly using a limited number of digits. (Story, 2004)

Finally, the digital representation of the sound wave is available as a stream of numbers for further processing. In the process, we have traded precision and accuracy for efficiency. However, further processing and transmission can be done losslessly.

3.2 Introduction to ADC systems

In different ADC systems, a variety of different methods exist for the determination of the sample values. One of the most common is the integration of the value over the sampling period. There are several other successful types of ADCs, including dual-slope converters (as illustrated in Fig. 3.5 and Fig. 3.6), residue converters, folding converters, parallel flash converters (as illustrated in Fig. 3.8), charge-redistribution converters (illustrated in Fig. 3.9), feedback-type converters, and oversampling noise shaped converters. Several types can be found as building blocks of a distinguished commercial ADC product. (Story, 2004)

3.2.1 Dual-slope ADC

A dual-slope converter and the underlying conversion scheme are illustrated on Fig. 3.5 and Fig. 3.6). Dual-slope converters are used in high-resolution products where conversion speed is not a crucial parameter.

The conversion consists of two phases. Initially, switch S_2 is closed to discharge the capacitor C. The switch is then opened and switch S_1 is connected to analog input signal v_A for a fixed time interval T_1, during which the digital counter counts pulses from a fixed-frequency clock. Usually the number of counted pulses, n_{REF}, is 2^N for an N-bit converter.

In the second phase, switch S_1 is thrown to the reference voltage V_{REF}. Again, pulses are sent to the counter logic. Total number of counted pulses n relates to the input voltage v_A:

$$n = n_{\mathrm{REF}} \left(\frac{v_A}{V_{\mathrm{REF}}} \right) \tag{3.1}$$

3.2.2 Feedback-type ADC

Feedback-type ADC is illustrated in Fig. 3.7. Instead of processing the input voltage directly, it employs a DAC, digital-to-analog converter, to create a matching analog signal level. Once the voltages match at the comparator, the binary word controlling the DAC can be read as a digital representation of the original input voltage v_A. According to Sedra and Smith (2004), the operation of a feedback-type converter, due to the up-down counter, is slow if started from zero, although incremental changes in the input signal are quite rapidly tracked. Moreover, the conversion time is dependent on the input signal level. For example, converting a single 16-bit audio sample would require (worst case) 2^{16} clock cycles.

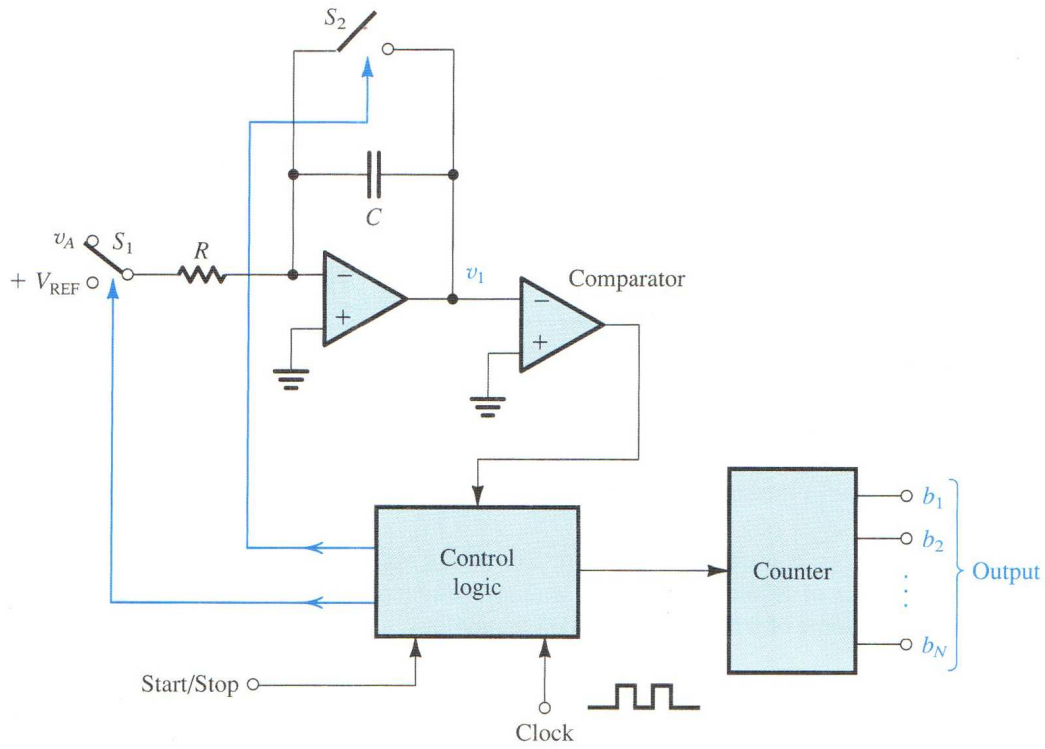

Figure 3.5: Dual-slope ADC. The voltage v_1 across the capacitor C is slowly charged and discharged during a variable intervals, depending on the analog input voltage v_A, while the control logic counts the pulses from a fixed-frequency clock. Input voltage thus becomes presented as a number of counted pulses that can be represented as a binary word. Adopted from Sedra and Smith (2004).

3.2.3 Flash ADC

Flash (or *parallel*) converters (as illustrated in Fig. 3.8) are – by design – fast, as all the bits of the digital word are created at once, but they require complex (in size) and expensive circuit implementations. The conversion process is conceptually straightforward. $2^N - 1$ comparators are used to compare the input signal with each possible quantization level at once in order to provide the N-bit word.

3.2.4 Charge-redistribution ADC

Charge-redistribution converters, as illustrated in Fig. 3.9, consist of a binary-weighted capacitor array, a comparator and set of analog switches. They operate in three phases. In sample phase the switch S_B is closed to connect the top plates of all capacitors to signal ground and switch S_A is connected to the analog input voltage v_A. Voltage v_A is then sampled and stored as a charge by the capacitor array.

In hold phase, the switch S_B is opened and switches at the bottom plates of capacitor

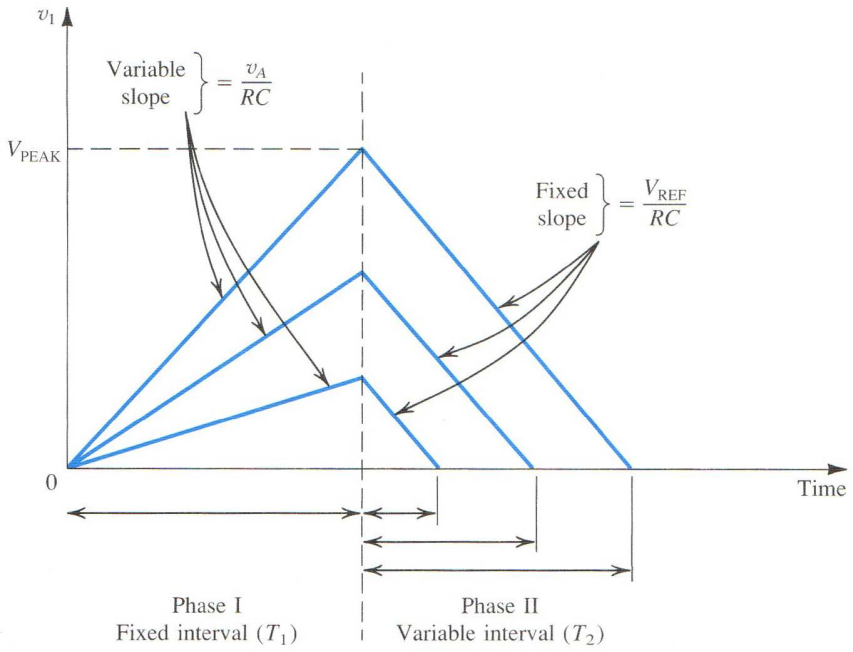

Figure 3.6: Dual-slope analog-to-digital conversion. The voltage v_1 across the capacitor C in Fig. 3.5 is slowly charged and discharged during a variable interval, depending on the analog input voltage v_A, while the control logic counts the pulses from a fixed-frequency clock. Adopted from Sedra and Smith (2004).

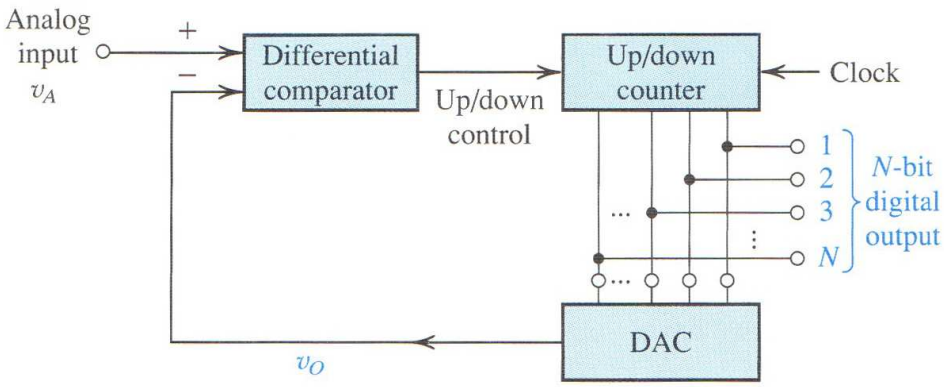

Figure 3.7: Feedback-type analog-to-digital converter. Adopted from Sedra and Smith (2004).

array thrown to ground, thus forcing the voltage at their top plates to $-v_A$.

In charge-redistribution phase, the bottom plate switches are tried to be thrown (as controlled by the comparator and control logic) to V_{REF} until the voltage on the top plates is reduced to zero, at which point the desired digital word is readable as the position of the switches.

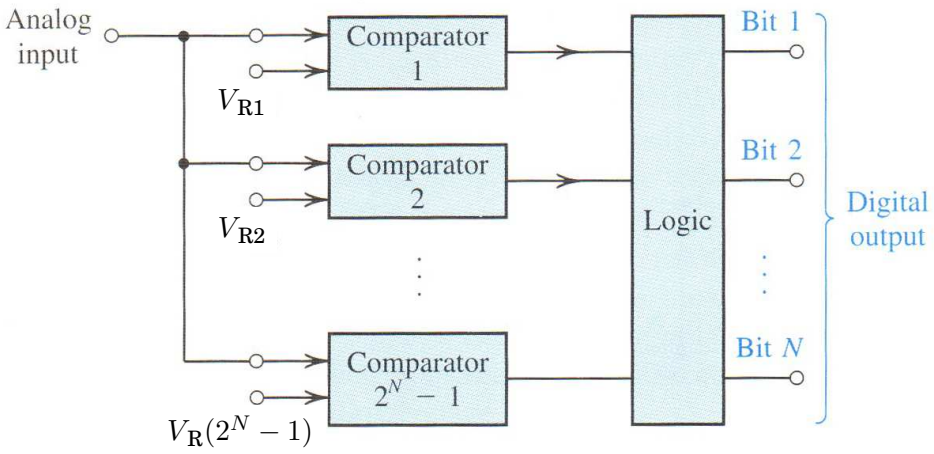

Figure 3.8: Parallel ADC. Analog staircase input signal from the sampler is sent to a set of parallel comparators, each weighted with exponentially increasing reference voltage. $2^N - 1$ comparators are used to compare the analog input signal to each possible quantization levels at once. Depending on the status of the comparators, digital logic circuit is used to switch appropriate set of output bits. Although not shown in the picture, the system is typically synchronized to a clock signal running at the desired sample rate. Adapted from Sedra and Smith (2004).

3.2.5 Limitations

It should be noted that the performance of an ADC is always limited. Nowadays, along with all the other factors, limiting the power consumption in SOC (system on a chip) implementations guarantees that compromises have been made. (Story, 2004) The theoretical SNR, a signal-to-noise ratio, related to the quantization noise, can be obtained from

$$SNR_{\text{dB}} = 20 \log_{10}(2^w), \tag{3.2}$$

that gives the ratio of the maximum signal value 2^{w-1} to the maximum quantization error. It follows that the dynamic range SNR_{dB} can be widened by increasing the number of bits allocated to sample bit depth w.

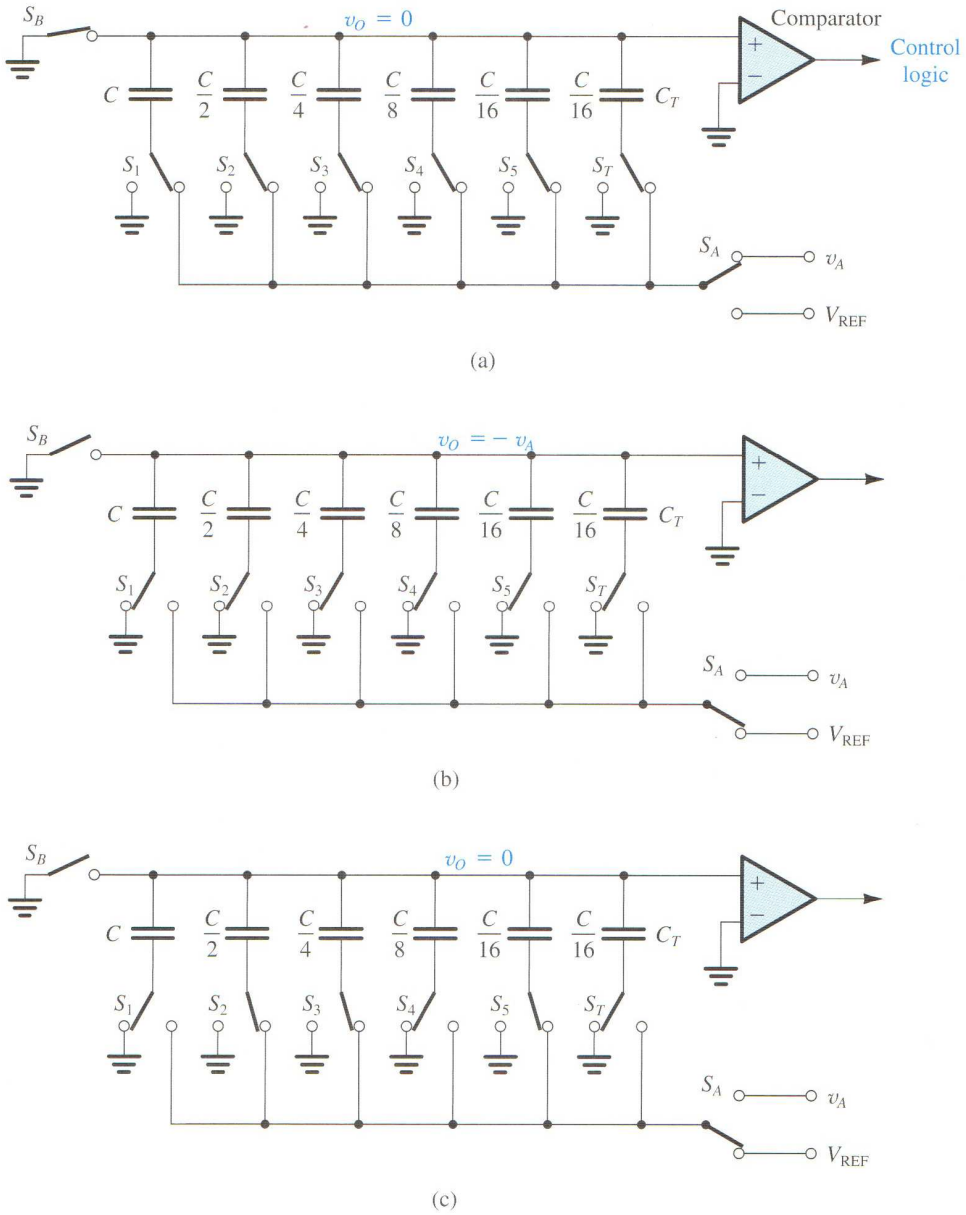

Figure 3.9: 5-bit charge-redistribution ADC. (a) Sample phase. The top plates of all capacitors are connected to ground. (b) During hold phase, the bottom plates of all capacitors are thrown to ground. As the system is open-circuited, the capacitor charges remain constant, thus $v_O = -v_A$. (c) Charge-redistribution phase. Adopted from Sedra and Smith (2004).

3.3 Dithering

Before the quantization stage, a dither noise signal is often deliberately added to randomize the quantization error, or more specifically, to prevent the error from being correlated to the signal itself. Omitting the dithering stage typically results in undesirable cyclical artifacts.

The dither signal can be added to the system at different entry points, in analog or digital format. The addition of dither linearizes the input-output characteristic, providing low-level resolution. The downside is that it also adds noise, which may be unacceptable.

Various analog dithering techniques have been analyzed by Vanderkooy and Lipshitz (1987). Useful analog noise signals have Gaussian or rectangular probability density functions. Binary noise can also be of use if its clock rate is at least three times as high as the sample rate. Optimal analog dither was found by Lipshitz and Vanderkooy (1986) to have triangular probability density function and a peak-to-peak amplitude equivalent to two least significant bits. The dither signal can be added before or after the anti-aliasing filter.

Lipshitz and Vanderkooy have also studied digital dithering. It differs from its analog counterpart mainly in that the wordlength is finite. A random binary number, below the binary point, is added to each digital sample before the truncation or rounding operation is performed. As many bits as possible below the binary point should be used in the dithering process.

Naus and Dijkmans (1988) have shown that the dither signal that should be added to prevent harmonic distortion can be obtained for free from thermal noise and 1/f-noise (also known as pink noise) from analog components by proper design.

3.4 Sigma-delta quantization in oversampling ADC

One very popular type of ADC is the sigma-delta converter. This type of converters use massive oversampling ratios with 1-bit ADC. In audio applications, the input signal is sampled with the rates of multiple megahertz. The structure of a sigma-delta converter is illustrated in Fig. 3.10. (Mitra, 2006)

The corresponding input and output waveforms of the abovementioned sigma-delta converter are illustrated in 3.11. It should be noted that the analog signal on the 3.11(a) can be obtained again from the quantized sigma-delta signal 3.11(b) via low-pass filtering.

Dunn and Sandler (1997) have studied psychoacoustically optimal sigma-delta modulation and how to make the noise floor minimally intrusive to the listener by using psychoacoustically optimal noise shaping instead of trying to yield maximal unweighted signal-to-noise ratio. Essentially, the quantizer noise is filtered to redistribute it to audio bands where it is less audible.

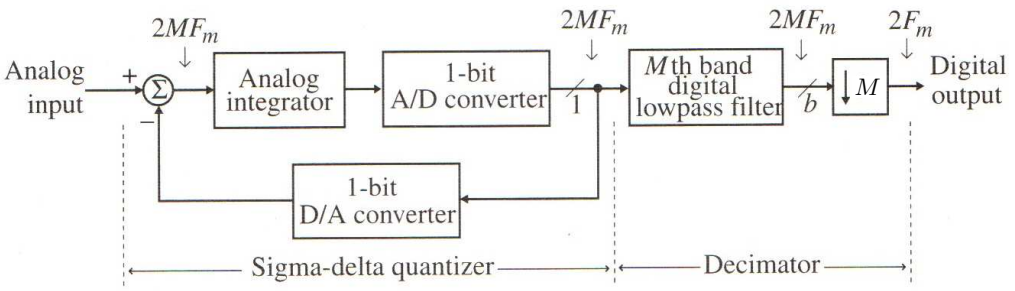

Figure 3.10: Oversampling sigma-delta ADC structure. M is the oversampling ratio and F_{m} the maximum frequency of interest in the input signal, the Nyquist frequency thus being $2F_{\mathrm{m}}$. Adapted from Mitra (2006).

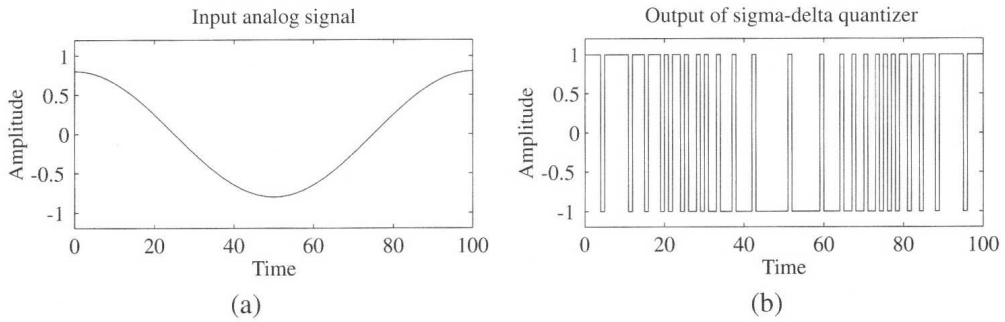

Figure 3.11: Input and output waverforms of a sigma-delta quantizer. The analog sine wave input is in (a) and the corresponding output is in (b). Adopted from Mitra (2006).

According to Angus (1999), signals could be processed in sigma-delta format directly, using one bit filter structures. Advantages over multi-bit processing were claimed to be naturally serial connections and component savings at the cost of faster clocking hardware requirements. Their findings were questioned by Lipshitz and Vanderkooy (2000), who showed that one bit converters overload and their operation becomes strikingly nonlinear if appropriately dithered. Multi-bit conversion was encouraged instead.

Angus (2001) later concluded that both multi-bit (PCM) and sigma-delta techniques are not only perfectible, but that they can realistically be compared. Overload distortion occurs when the step size in the sigma-delta modulator is too small and the modulator output thus cannot follow the analog input fast enough. If on the other hand the step is too large and the input signal is slowly varying, excessive granular noise dependent on the input waveform will occur. Reefman and Janssen (2002) showed that the distortion in sigma-delta conversion can be greatly improved by their sigma-delta pre-correction technique.

Reiss (2008) has provided, along with a great sum-up of delta-sigma techniques, a convenient comparison for oversampling and bit depth. SNR improves by 6 dB with each extra bit, but only 3 dB by doubling the oversampling ratio. Thus the equivalent SNR of 16 bits at 44100 kHz using a 8-bit quantizer would require a sample rate of nearly 3 GHz.

Additionally, more practical applications for sigma-delta modulation have been found recently. While studying entirely digitally driven speakers, Ogata et al. (2006) proposed using sigma-delta modulation in the speaker element. Extensive studies of this topic have been then made by Kuroki et al. (2008), Watanabe et al. (2009) and Kuniyoshi et al. (2010).

3.5 Introduction to DAC systems

Conversely to the ADCs, a digital-to-analog converter, or DAC, accepts an N-bit digital word and produces an analog sample. As a concept, DAC is much less complex, as there is a finite number of possible states of the digital input word. Essentially a DAC consists of a staircase signal generator connected to a suitable lowpass filter. Adaptation from the Reeves (1942) original PCM patent illustration in Fig. 3.12 provides a practical example of an early 5-bit DAC. The binary input word is first loaded into the counter, and the set-reset flip-flop (FF) is reset. The counter is then allowed to count upward by applying 6 kHz clock pulses. When the counter overflows and reaches 00000, the clock source is disconnected, and the flip-flop is set. The number of pulses counted is the complement of the digital input. The output of the flip-flop is a pulse-width modulated signal whose analog value is the complement of the input word. A lowpass filter (LPF) is used to recover the analog signal.

A better understanding about the electrical implementation can be obtained by observing the N-bit DAC by Sedra and Smith (2004) in Fig. 3.13. Exact details are beyond the scope of this thesis. The DAC uses a binary-weighted resistive ladder network. The switches (noted S_N in the figure) open and close based on the digital input word D on the input. There are two grounds in the picture. Position 1 of the switches is the real ground and position 2 is the virtual ground, so that the currents through the binary weighted resistors remain constant. Currents to the virtual ground add up and flow through the feedback resistor R_f, generating a voltage $v_0 = -i_0 R_f = -V_{\text{REF}} D$. Each bit is thus connected to the correspondingly (exponentially) weighted resistor, through which the current flows, resulting in an output voltage that is the analog representation of the input word. This, however, is not the same as the desired analog signal, but a staircase waveform. According to Mitra (2006), the analog signal can be recovered from the staircase waveform via lowpass filtering.

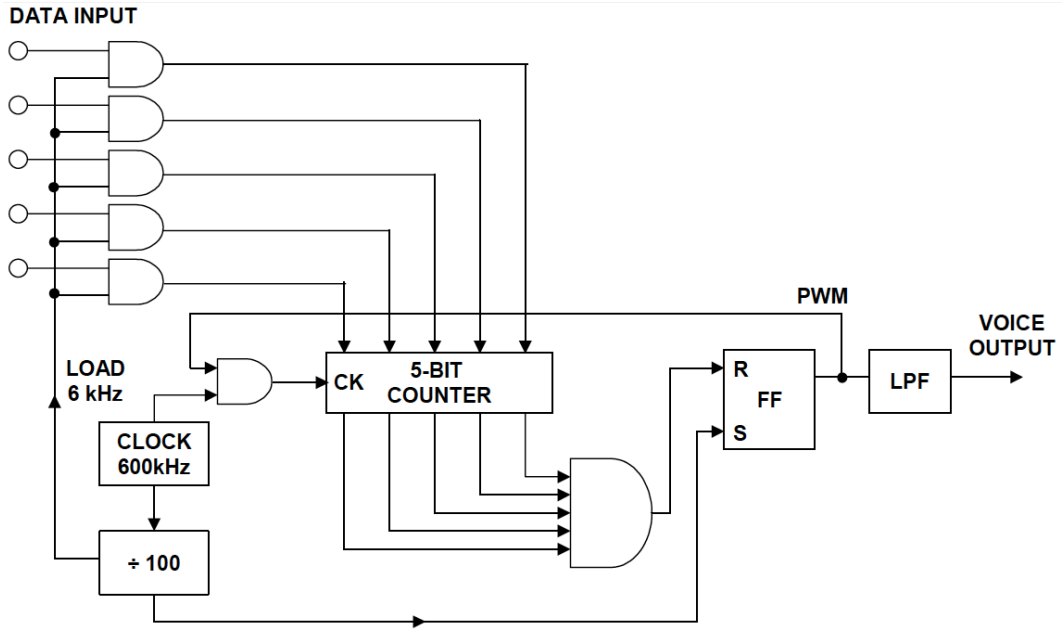

Figure 3.12: Adaptation by Kester and Analog Devices Inc. (2005) of the 5-bit counting DAC of the (Reeves, 1942) patent.

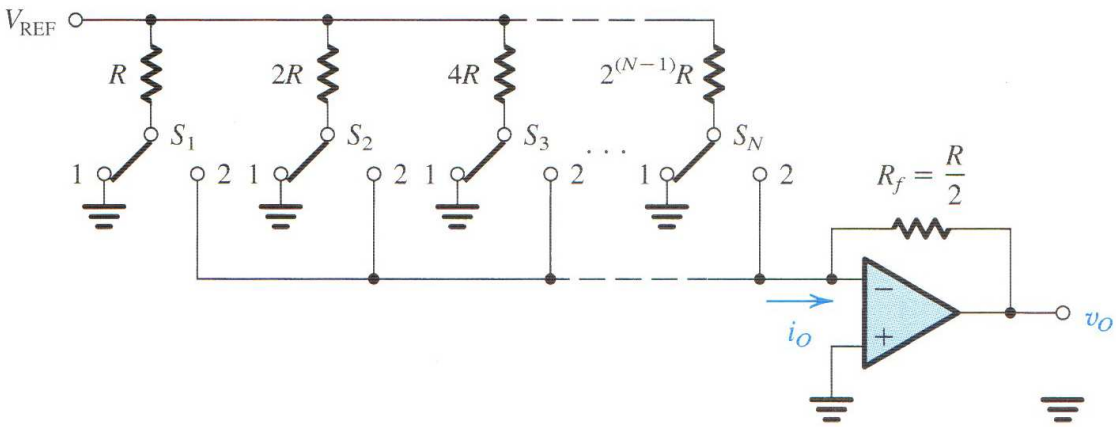

Figure 3.13: N-bit DAC using a binary-weighted resistive ladder network. Switches S_N are controlled by the digital input word D. There are two grounds in the picture. Position 1 of the switches is the "real" ground. Position 2 is the virtual ground, so that the currents through the binary weighted resistors remain constant. Currents to the virtual ground add up and flow through the feedback resistor R_f, generating a voltage $v_0 = -i_0 R_f = -V_{\text{REF}} D$. Adopted from Sedra and Smith (2004).

Chapter 4

Introduction to digital audio programming

In this chapter, a motivational introduction to audio programming is given. Relation of source code and processor instructions is discussed briefly. Additionally, a digital filter is explained by an example.

4.1 Motivation to digital audio programming

For any communication to succeed, parties on both ends must agree on the interpretation of the data being transmitted. At a higher abstraction level in digital communications, the parties trivially agree on the status of isolated bits being either zero or one as this is handled by lower level components. Yet, challenges lie in consistent interpretation of the order of bytes, samples, and channels.

A fundamental property of digital circuits is the time-delay of one sampling period T_S. It is the inverse of operating frequency of the device as shown in Eq. 4.1, and represents the minimum possible delay of operation. In other words, the duration of one sample in time, or the time between samples, (T_S) is the inverse of sampling frequency f_S. The operation of digital circuits can be described as a network of digital delays and signal gains.

$$T_\mathrm{S} = \frac{1}{f_\mathrm{S}} \tag{4.1}$$

Practically all sound devices employ some kind of buffering. As even the slightest loss of data is instantly noticeable on audio stream playback, this situation is avoided by adding various safety margins which in turn add to total latency. The most common

is to use two buffers, one of which is read by the device while the other is being written into by the host. This approach is commonly called double buffering. In other fields of computer sciences, such as in real-time three-dimensional graphics, even triple buffering is often used. Multiple buffers enable the use of more complex effects that require memory of past frames, for example a full screen motion blur.

4.2 From source code to instructions

As explained by Stroustrup (2005), although most applications have code sections critical to the performance, the largest amount of code is in other sections. Low-level efficiency enables one to write software that manipulates hardware under real-time constraints. Stroustrup emphasizes the predictability of performance over raw speed. For most code, however, maintainability and ease of extension and testing outweigh speed. Widely known and long-established application programming interfaces (APIs) fill the gap by providing the application programmer tools that both perform predictably and are optimized for speed.

Before a C language application can be executed, it has to be *compiled*, as illustrated in Fig. 4.1. The compiler is a special application that generates an executable binary based on the given application source code. A compilation process of an application has roughly two parts. First, the source code and all the necessary library headers are processed to create what is called an intermediate presentation. The source code is also parsed for syntax errors. Second, a system hardware specific assembly code is generated that is *linked* to the hardware specific libraries provided by the API manufacturer.

When the compiled audio application is executed, data and operatives flow as illustrated in Fig 4.2. As the application “speaks” to the underlying hardware, the message is translated multiple times. Kernel in turn operates with the firmware, a highly specialized small operating system running in the sound card CPU (sometimes called APU, the audio processing unit) that controls the audio hardware, namely the DAC.

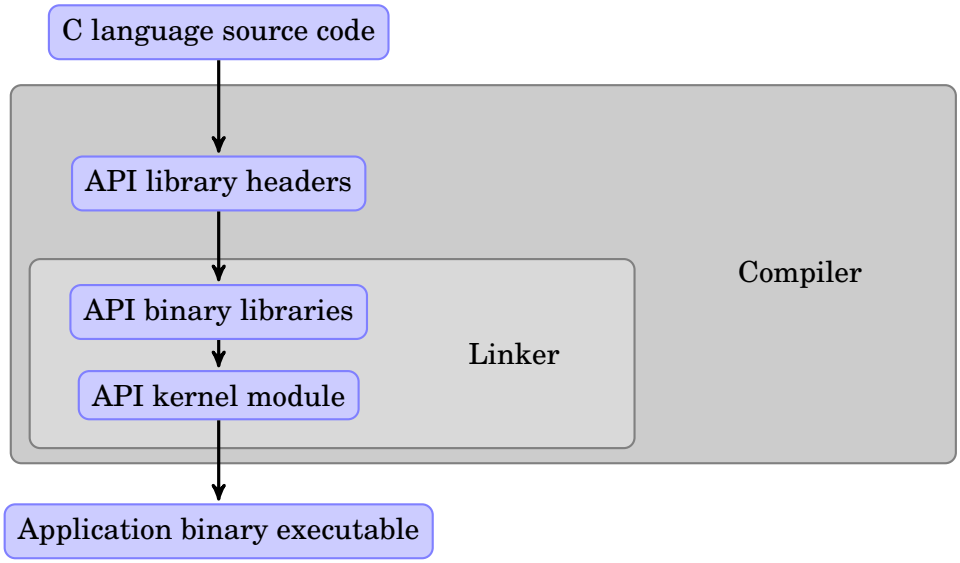

Figure 4.1: Compiling an audio application. Source code is first combined with API headers then linked to API binary and API kernel libraries, thus generating a hardware-dependent executable.

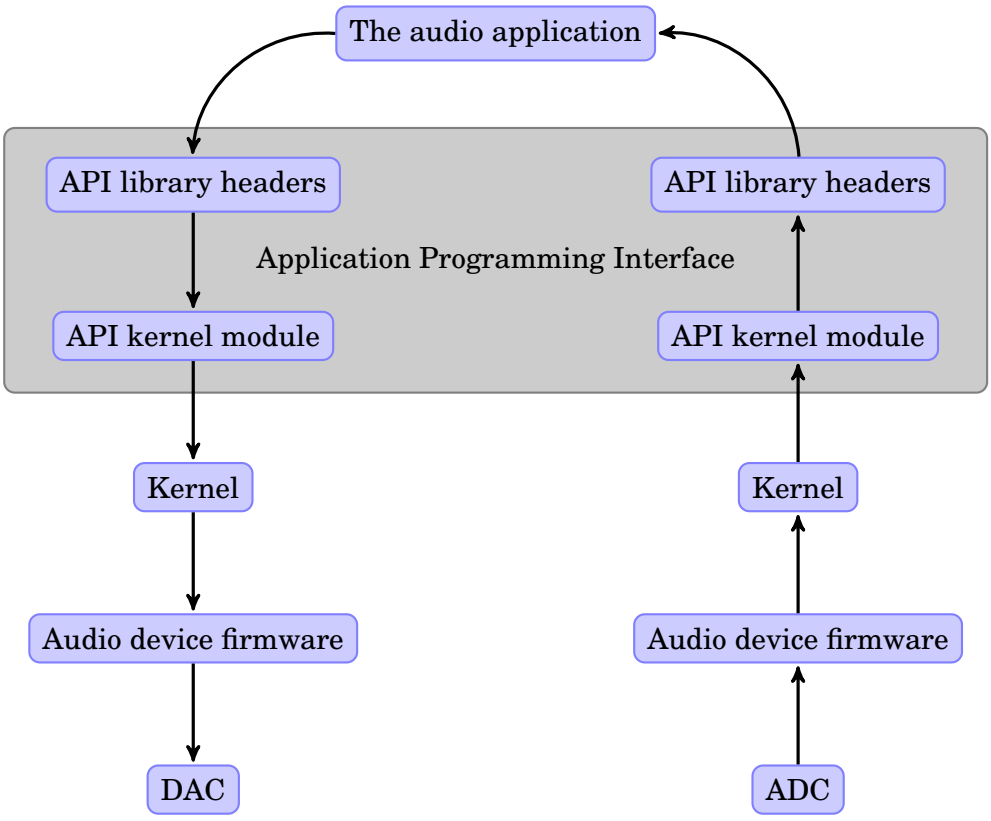

Figure 4.2: Flow of data in a real-time I/O audio system

4.3 A digital two-point moving-average filter

A moving-average filter is perhaps the simplest-to-implement finite impulse response lowpass digital filter. It is a first-order filter structure commonly called an accumulator. Its difference equation in causal format is

$$y[n] = \frac{1}{N+1}\left(x[n] + x[n-1] + ... + x[n-N]\right). \tag{4.2}$$

According to Mitra (2006), one of its popular applications is in estimating data vectors by taking ensemble average from multiple measurements of noise-corrupted data samples. The difference equation of its first-order version, the two-point moving-average filter, is

$$y[n] = \frac{1}{2}\left(x[n] + x[n-1]\right). \tag{4.3}$$

The magnitude response of such a filter can be derived by using the z-transform to obtain

$$Y(z) = \frac{1}{2}X(z)z^0 + \frac{1}{2}X(z)z^{-1}, \tag{4.4}$$

and finally dividing this by $X(z)$ to obtain

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{2}z^0 + \frac{1}{2}z^{-1}, \tag{4.5}$$

the magnitude response. As we have the identities

$$z = e^{j\omega} = e^{j2\pi f}, \tag{4.6}$$

and

$$z^0 = 1, \tag{4.7}$$

where the z (transform) operator stands for the delay in the discrete time sequence (Hirata, 1981), the frequency response of the filter

$$H(e^{j2\pi\frac{f}{f_\mathrm{S}}}) = \frac{1}{2} + \frac{1}{2}e^{-j2\pi\frac{f}{f_\mathrm{S}}}, \tag{4.8}$$

where f_S is the sample rate, can be obtained from Eq. 4.2.

The magnitude response $|H|$ as a function of angular (normalized) frequency is

$$\left|H(e^{j2\pi\frac{f}{f_\mathrm{S}}})\right| = \left|\frac{1}{2} + \frac{1}{2}e^{-j2\pi\frac{f}{f_\mathrm{S}}}\right|. \tag{4.9}$$

At cutoff frequency, the magnitude response of the filter drops below -3 dB. Eq. 4.9 at cutoff frequency f_C becomes

$$\left|H(e^{j2\pi\frac{f_\mathrm{C}}{f_\mathrm{S}}})\right| = \left|\frac{1}{2} + \frac{1}{2}e^{-j2\pi\frac{f_\mathrm{C}}{f_\mathrm{S}}}\right| = \sqrt{\frac{1}{2}} \approx -3 \text{ dB}. \tag{4.10}$$

It should be noted that the response of the digital filter to one particular frequency depends on the sample rate. Changing the sample rate f_S also changes the cutoff frequency f_C of the (same) filter structure.

By setting $|H| = \sqrt{1/2}$ and rewriting Eq. 4.10 as

$$\sqrt{\left(\frac{1}{2} + \frac{1}{2}e^{j2\pi\frac{f_\mathrm{C}}{f_\mathrm{S}}}\right) \cdot \left(\frac{1}{2} + \frac{1}{2}e^{-j2\pi\frac{f_\mathrm{C}}{f_\mathrm{S}}}\right)} = \sqrt{\frac{1}{2}}, \tag{4.11}$$

which can be reduced to

$$\frac{1}{4} + \frac{1}{4} + e^{j2\pi\frac{f_C}{f_S}} + e^{-j2\pi\frac{f_C}{f_S}} = \frac{1}{2}, \tag{4.12}$$

and further reduced to

$$e^{j2\pi\frac{f_C}{f_S}} + e^{-j2\pi\frac{f_C}{f_S}} = 0, \tag{4.13}$$

which can (according to e.g. Hutchins et al. (1975)) be reinterpreted as a trigonometric equation using Euler's identity,

$$e^{i\theta} = \cos(\theta) + i\sin(\theta), \tag{4.14}$$

to obtain

$$2\cos\left(2\pi\frac{f_C}{f_S}\right) = 0, \tag{4.15}$$

which can be reduced to

$$2\pi\frac{f_C}{f_S} = \frac{\pi}{2}, \tag{4.16}$$

and finally to

$$f_{\mathrm{C}} = \frac{f_{\mathrm{S}}}{4}, \tag{4.17}$$

from which the relation between the sample rate and the cutoff frequency of a two-point moving-average filter can be obtained to get the frequencies listed in Table 4.1.

Table 4.1: Some cutoff frequencies of the two-point moving-average filter

Sampling frequency / Hz	**-3 dB cutoff frequency / Hz**
44100	11033
48000	12000
96000	24000

Chapter 5

Audio programming interfaces

In this chapter, some properties of application programming interfaces for audio applications are explored. The reasoning behind the selection of the programming interfaces for this thesis is discussed. Additionally, introductions of the selected interfaces are given.

5.1 Evaluation of API usability

Nowadays, application programmers do not have to know the quirks of individual hardware devices. Instead, they have to cope with large code libraries and frameworks, with numerous classes and methods. As majority of the classes and methods in the APIs are meant for widespread use, their documentation is written so that everything a user might need to know is included. This has the unfortunate consequence that most of the documentation is not required by any given single programming task and in fact slows down the development phase as the programmer is forced to read through material that is non-relevant, probably missing important information about usage directives concerning particular method invocations. Some of these problems can be addressed inside integrated development environments (IDEs) by means of *knowledge pushing*, as shown by Dekel and Herbsleb (2009).

Common API learning obstacles have been studied by Robillard (2009). According to the study, developers need information about the high-level design of the API in order to choose between alternative ways of using it. Code examples were found to be useful only when they happen to accord with the exact purpose of the programmer. Similar observation was made during this study. MSDN (Microsoft Developer Network) WASAPI documentation includes separate code examples for real-time input and real-time output. However, a programmer seeking information about simultaneous I/O might get confused by the two examples as they lack vital information about copying data from one buffer to another and about avoiding buffer underflows.

Complexity of the API increases the amount of time taken to find classes and methods

required for a specific task. This, however, is quickly compensated by the learning effect if the same programmers can apply the same API multiple times. According to Scheller and Kuhn (2012), the search and instantiation times of API components is cut considerably during the first few times the programmer applies it for a task, improving performance. When using a new API for the first time, a lot of time is spent on finding the right class, depending on the amount of possible extra help from the IDE used.

There are other important selection criteria not thoroughly studied in this thesis. Although all three APIs selected for this thesis are freely available, due to them being tied to the underlying operating systems they are not entirely free. A typical consumer has already selected a certain device platform and the operating system that goes with it, which reduces the freedom of choice of the API.

Rather surprisingly, of the studied APIs in this thesis, not one covers simultaneous I/O in the official documentation. The official C library reference of ALSA project offers developers with a total of five example programs. There are working examples of real-time output with a variety of different technical approaches, but no help is given for combining this with input. Microsoft MSDN offers excellent documentation with working example code for rendering a stream and capturing a stream, respectively, but no example is given for doing both at the same time. Apple encourages the use of the so-called Audio Units. Another approach is to use the lower level Core Audio framework directly, as is done in the work for this thesis. The individual methods for input and output applications are well documented on the Mac Developer Library website, but an example code has to be sought elsewhere.

5.2 Integrated Development Environments

Some of the API complexity can be mediated by using an integrated development environment, IDE. IDE is a task-special user interface with shortcuts to necessary tools like automatic linking and compiling of the program. Often the word IDE is associated with graphical user interfaces, but IDE can be text-based. One could think of the operating system shell as an IDE, too.

Essentially, the IDE can be as elementary as a text editor that can interpret the basic structure of the code and fix some grammatical errors for the programmer. A common feature is the automatic color-coding of different parts of the program source code which helps by visualizing the code and assists the programmer, for example, to immediately see which parts of the source code have been commented out.

More advanced features may include a method attribute helper that warns about trying to push floating point value to a method that expects integers, even before the programmer tries to compile the source code, and even offers to fix the problem auto-

matically.

5.3 Choosing the right APIs

When planning an audio application, a decision must be made on what APIs to utilize. This thesis studies the real-time performance of an application. Minimizing the latency has been prioritized over other factors. There are numerous APIs available. In order to achieve low latency, one should avoid using unnecessary components in the signal chain. Decision-making becomes easier when one observes that many of the APIs are built on top of the others.

The question of proper hardware support naturally dominates the API selection process. The API has support all the targeted hardware. The laptop that was available for this thesis could effortlessly run three popular operating systems. In turn, a variety of APIs support these operating systems. First, the operating systems themselves provide individual low-level audio APIs that the creators of the operating system believe to suit most users. Mac OS X offers the Core Audio (detailed in Section 5.4), Windows provides the Windows Audio Session API (WASAPI, detailed in Section 5.5), and most Linux distributions include the Advanced Linux Sound System (ALSA, detailed in Section 5.6). These are closely tied to the underlying operating system and are primarily meant as a backend for more user-friendly and complex APIs, some of which are now briefly studied.

PulseAudio is a cross-platform network sound server project initiated by Lennart Poettering in 2004. It does software mixing, networked audio (with which one can utilize audio hardware on multiple computers) and furthermore, hardware abstraction. Virtualized soundcards enable user-friendly features like per-application volume control. For hardware input and output, PulseAudio uses ALSA, WASAPI, Core Audio, JACK, and others depending on the platform it is run at. If the application only needs to handle audio streams to and from audio hardware, it does not need PulseAudio. The application can access the audio backends directly, thus replacing PulseAudio. (PulseAudio Developers, 2013)

JACK Audio Connection Kit can use ALSA, Core Audio or WASAPI as its backend, but also a variety of others, including PortAudio. (Davis, 2011) Although JACK is built to handle “real-time low latency audio”, it can only offer as low latency as its backend can. If the application would utilize only a single PCM input and output, using JACK has latencywise no advantage over using the backend directly.

PortAudio Portable Real-Time Audio Library is a cross-platform audio library by Ross Bencina and Phil Burk, that can use, among others, ALSA, Core Audio, JACK, or WASAPI, as its backend. PortAudio is free and open-source. (PortAudio Community, 2013)

Some of the above also mention DirectSound support in Windows platform. DirectSound, or DirectX Audio, is a higher level API for Windows multimedia services. As stated by Microsoft Corporation (2012b), DirectSound uses (Windows) Core Audio APIs, such as WASAPI, as its foundation. Therefore, unless higher level services, such as hardware accelerated MPEG codecs, or DRM (digital rights management), are required, there is no reason to use DirectSound.

Instead of using the backend libraries, one could rewrite them to include the desired application code as well, thus creating ones own backend. In embedded solutions this might be feasible. Obvious drawback is that the application then becomes hardware dependent.

It can be concluded, that of the APIs mentioned above, those depending on others as their backend cannot provide any latency advantage over using their backends directly. As the ultimate objective is to minimize the latency, for the given hardware platform, only three of the API candidates are relevant to be studied further. In this thesis, the audio programming interfaces listed on Table 5.1 are explored. Performance of each API in terms of combined I/O latency is analyzed by implementing a Karplus-Strong plucked string synthesis patch to be run on the same hardware. A real-world, real-time performance is analyzed.

The programming language chosen for all test applications used in this thesis is C. C is, arguably, the most popular programming language on the planet Earth. C language is source code level compatible with all the three APIs, ALSA, Core Audio and WASAPI, as it can be used alongside objective-C and C++ languages without source code compatibility wrappers.

Table 5.1: List of audio programming interfaces explored in this thesis

Interface	**Platform**	**API language**
ALSA	Linux	C99
Core Audio	Mac	Objective-C
WASAPI	Windows	Visual C++

5.4 Core Audio

The Core Audio API services visible to the programmer are layers on top of HAL, the Hardware Abstraction Layer. HAL masks the hardware and drivers and instead provides the programmer with a consistent interface independent of the actual hardware. All the audio devices in the system can be found simply by querying the simpleton `AudioSystemObject` which serves as the root of the device hierarchy.

In order to utilize Core Audio, the application programmer is only required to include the framework headers, namely `CoreAudio.h` and `CoreFoundation.h`. These in turn include all the necessary headers for the framework. Apple also provides a special version of the C compiler that takes the name of the framework as a parameter instead of individual libraries, simplifying the build process, but also breaking the compatibility and portability with other platforms. It is possible, though, to create a multiplatform makefile system using external `uname` shell command and `ifeq` conditionals.

Audio I/O operations in Core Audio are based on callbacks. The operating system handles the audio devices frame buffers, so that individual applications can only see memory pointers to the virtual audio streams provided by the operating system. When an application wants to interact with these audio streams, it must register function pointers to its own audio handling functions. The operating system has a list of functions it takes care of calling whenever there is a buffer full of data to be read or written for a particular device. Therefore, multiple applications can share the devices.

For customers not particularly interested in programming finesse, an API with a steep learning curve is useless. The main header files of Core Audio Framework are filled with appropriate comment blocks and each function parameter and magic number is thoroughly described. Parameter and function names are often self-explanatory. Both hardware and operating system compatibility of audio functions are listed in header files.

In addition to developer resources, some parts of the Mac OS X and iOS operating systems (for example, `IOAudioFamily`) are released as open source on Apple website Apple Inc. (2013). Complete Core Audio framework source code is not included, though.

Individual devices in Core Audio operate asynchronously by default. In particular, there is no guarantee that subdevices of even the same hardware would be synchronized. This has a dramatic effect on, for example, latency measurements, as the amount of delay caused by internal buffering can change freely between device restarts, and could change also while the devices are running.

A practical way to ensure device synchronization in Core Audio is via the *aggregate devices*. Individual hardware devices can be combined within the operating system into virtual devices, that take their clock input from one of the subdevices, and the others are synchronized to that. The operating system ensures that the hardware buffers are kept in sync.

No easily achievable exclusive sound system access mode was found in Core Audio. However, the buffer size for the device can be altered from within the application. The default buffer size in Core Audio for the hardware used in the measurements

was found to be 512 frames.

`kAudioDevicePropertyBufferFrameSizeRange.mMinimum` can be probed by the `AudioObjectGetPropertyData` method to find the minimum allowed frame size limitation for a particular device, in order to, achieve even lower latency in a Core Audio application.

Afterwards, `kAudioDevicePropertyBufferFrameSize` can be used to set the number of frames in the I/O buffers. Then again, one must monitor the results for possible buffer overruns or underruns that are due to happen if buffer size restrictions are set too tight. (Apple Inc., 2006)

During the measurement runs, if there was a buffer underrun or overrun detected, the process was halted and restarted. Using a buffer size of 64 frames, there were a few incidents. Due to the nature of the multitasking general-purpose platform, a number of buffer underruns and overruns is expected as some operating system kernel operations have priority over the audio applications and do occasionally interrupt the process. (Bencina, 2011)

5.5 Windows Audio Session API

The Windows Audio Session API (WASAPI) is the interface that client applications in Windows environment use to exchange and manipulate the audio data of audio endpoint devices. It separates the user mode audio applications and kernel mode audio drivers and actual audio hardware beneath the abstraction layer. Higher level Windows audio interfaces like the DirectSound and Windows Multimedia are layers on top of WASAPI. In Microsoft terminology, WASAPI is, quite confusingly, often referred to as one of the *Core Audio* APIs. (Microsoft Corporation, 2012b)

WASAPI is accompanied by MMDevice, or The Windows Multimedia Device API, that enables the programmer to discover the various endpoint devices in the system and study their capabilities in order to determine if the desired devices for the task at hand are available.

WASAPI is defined by two main header files, `Audioclient.h` and `audiopolicy.h`. In addition to these, a working WASAPI application has to include additional headers like `MMdeviceapi.h` for device discovery, `ObjBase.h` for `CoInitializeEx` function mandatory for the device enumeration. `MMReg.h` defines the various multimedia formats recognized by the Windows operating system and is in practice mandatory for a working application. In order to extract the human readable names of the audio devices to be shown to the end user in the user interface at run time, the magic numbers defined in `FunctionDiscoveryKeys_devpkey.h` have to be included.

Compared to the Apple Core Audio, where the application programmer only needs to include two foundation headers, the total amount of included Windows audio headers is six. Most of these requirements were found by the author by trial and error during the test application development process.

The `IAudioClient::Initialize` method of the WASAPI audio client takes the share mode as an argument. In order to open access to the device in shared mode the method must be called with `AUDCLNT_SHAREMODE_SHARED` as the first parameter.

In the WASAPI measurement main loop, the `IAudioCaptureClient::GetBuffer` method is used to obtain an `UINT32` value called `pNumFramesToRead`. This gives us the number of frames available to be read in the input device. In Microsoft terminology this group of frames is referred as a packet of data. According to WASAPI documentation, the client is then expected to read and process the whole packet or none of it.

By calling the `IAudioClient::Initialize` method with `AUDCLNT_SHAREMODE_EXCLUSIVE` as the first parameter, Exclusive access to WASAPI device can be requested. Contrary to the shared mode, the application must now submit a sample format proposal (typically in `WAVEFORMATEXTENSIBLE*` structure) in terms of the desired sample format, the sample rate, the number of audio channels and the byte alignment information.

In WASAPI measurement main loop, `IAudioCaptureClient::GetBuffer` method is used to obtain UINT32 value called pNumFramesToRead. This gives us the number of frames available to be read in the input device. In Microsoft terminology this group of frames is referred as a packet of data. According to WASAPI documentation, the client is then expected to read and process the whole packet or none of it.

The default buffer size Windows uses for the measurement setup was found to be 4416 frames. According to MSDN documentation, Intel HD Audio specification mandates that the length of the I/O buffer has to be a multiple of 128 bytes. In these 128 bytes, exactly 32 16-bit stereo PCM frames can be stored. Starting from Windows 7 (and Server 2008 R2, which has the same kernel), there is an upper limit for the size of the buffer, equivalent of 500 milliseconds. (Intel Corporation, 2010)

Usually, the buffer size in audio systems is expressed as the number of frames, or the number of samples that can be fit into it. In WASAPI, the buffer size is an unsigned integer type, as might be expected, but the unit is one hundred nanoseconds, or a $10^{7\ \text{th}}$ of a second.

5.6 Advanced Linux Sound Architecture

The ALSA project was initiated by Jaroslav Kysela in the 90s. ALSA evolved from a Linux device driver for the Gravis Ultrasound card to a complete audio solution.

ALSA code was integrated into the Linux kernel in 2002, eventually replacing Open Sound System as the uniform Linux sound API.

ALSA consists of three distinct entities. First, are ALSA kernel drivers, that are actually an integral part of the Linux kernel tree. Second, is a userspace library that enables applications to use the API methods to process sound. Userspace components complement the kernel drivers and add some high-level functionality, including software volume control and mixing. Third, ALSA is a collection of utilities like mixer controls and soundcard configuration tools. In addition to audio, ALSA provides MIDI functionality.

For a while in the beginning of the 21^{st} century, ALSA was the de facto Linux audio interface. Lately, it has been gradually being replaced by the Pulseaudio framework as the front-end, although Pulseaudio itself uses ALSA as its audio hardware back-end.

ALSA is defined by the `asoundlib.h` header. Enumeration of devices in the ALSA realm begins with the `snd_device_name_hint` function that takes three parameters. First parameter can be used to list the physical sound cards, or replaced with a magic number -1, which instructs the function to return devices from all available physical interfaces as a `char*` array.

The ALSA implementation of the main audio device structure is quite peculiar, as all the devices are accessed via their string literal names. For example, the string `hw:CARD=M2496,DEV=0` might stand for the first hardware PCM subdevice of an M-Audio Audiophile soundcard, but no assumptions should be made based on the name alone, as it can be an alias for something else.

In order to analyze the capabilities of different devices, `snd_device_name_get_hint` function can be used. The most important is the `IOID` property, that can be used to determine, whether the subdevice is capable of input, output, or both (in which case the value of the parameter is quite illogically NULL).

In ALSA, the programmer is expected to either poll the device for available buffer length. Alternatively, the client application can ignore the buffer length entirely and push as much data to as it pleases. ALSA then handles chopping and transferring the data appropriately. This has the obvious disadvantage that if the device buffer was full already it might take a long time for the function to finish.

In order to accomplish the ALSA measurements made in this thesis, an alternative measurement solution is proposed. The sole purpose of the application is to transfer data from input device to output device. Additionally, the input device can be queried about the amount of data there is ready to be collected. This data is read from the device using the `snd_pcm_readi` command. The output device is then queried about

the number of free sample slots on its ring buffer. There is no sample rate or sample bit depth conversions taking place so the buffer lengths can be compared directly.

Due to automatic conversions, usage of software devices is effortless. The application can choose almost any combination of a sample rate and buffer size. The selection does not affect the properties of the underlying hardware device.

Only as much data can be read from the input device as there happens to be ready. However, it is seldom necessary to settle for the initial number of frames ready at the input buffer. Instead, the read process can wait additional clock cycles for more data to appear. Initial read from the input sets the minimum achievable delay as there is no way to push data to output faster than real-time. If the capture of the first block of input data is started at time t_0 and its transfer to host is completed at t_1, the system has to have a delay of at least $t_1 - t_0$.

In ALSA terminology, one period is the size of the host I/O buffer that is copied to or from the ring buffer of the audio hardware as a whole. One period is therefore the minimum possible blocksize in ALSA. The size of this block typically can be set to as low as 64 samples, therefore implying that at 44100 Hz sample rate, the system must have a delay of at least 1.5 ms. The optimal initial read depends on the hardware and configuration. For measurement hardware used on this thesis, it was found by trial and error that when using a period size of 64 frames and number of periods on the buffer was three, that the initial read of 160 frames would suffice so that there would not be a buffer overrun nor underrun.

Chapter 6

Audio program in a nutshell

In this chapter, a schematic description of a generalized real-time audio measurement application is given.

6.1 Device selection

Firstly, the API has to initialize itself, reserve some memory and initialize some constants for the application. Some APIs can omit this phase as they can rely on the fact that the sound system is already initialized.

The second logical step of any program willing to utilize audio is to check the availability of sound devices in the system. After all, it is a possible scenario that there are no devices installed with working device drivers. This situation is of course acceptable if the application can notify the user and possibly wait for devices to become available. Modern systems often have multiple audio devices. The end user often wants to choose between Analog PCM stereo, coaxial or optical S/PDIF, Dolby Digital, DTS, HDMI, so these options somehow have to be evaluated by the application.

After device enumeration, there is an optional interactive phase where the end user is presented with a list of suitable audio devices present on the system. After user has chosen the appropriate device, the application must check that the device is apt for the task ahead. Otherwise, there should be a program termination or a forced reselection of the device.

Alternatively, an application may allow the user to select devices using command line parameters, or a configuration file. In any case, care should be taken that the previously selected device still exists on the system. If no audio device is found, the application should either exit or wait for additional devices to be connected.

6.2 Device configuration

As the device is now chosen, the next step is to extract sample rate and data format information, and depending on the task, to set these variables for the device. Additionally, if there is a mismatch between device sample rates that cannot be solved by changing settings, additional sample rate conversion code must be initialized in the application. A similar approach has to be taken if the audio data format is not constant throughout the system.

Additional care should be taken in cases where multiple channels are mixed into the same data stream so that samples from consecutive channels are properly combined into an interleaved stream to maintain channel order. In rare cases, interleaving of channels is done using larger block sizes.

A selection of the access mode is often done at this point, if applicable. Devices can usually be shared between other applications in the system, or the application can request exclusive access (which may or may not be granted). Shared access typically limits the application's degrees of freedom in terms of sample rate, buffer length, and stream data format.

6.3 Device start

Before any device can be read and written to, the API typically requires the application to issue a launch command before the device I/O streams become available. In ALSA, this function is called `snd_pcm_start`, the WASAPI method being `IAudioClient::Start`. In Apple Core Audio the same function is `IAudioDeviceStart`.

Devices should not be started carelessly though. For example, starting an output device when there is no data in the output buffer leads to immediate buffer underrun anomaly. Similarly, there has to be room in the buffer used by the input device for writing when it is started, or buffer overrun will occur. According to Davis (2002), "you can also start the device explicitly using `snd_pcm_start`, but this requires buffer prefilling in the case of the playback stream. If you attempt to start the stream without doing this, you will get -EPIPE as a return code, indicating that there is no data waiting to deliver to the playback hardware buffer."

As we are doing input and output, we typically have at least one device for input stream, and at least one other device for output. These devices can be separate physical devices. Typically they are different virtual devices of the same sound card. More importantly, only one of them can be started first. Others start at some other time. Therefore, even in the straightforward case involving a single input and a single output device, the devices are running asynchronously. This has a dramatic effect on

latency between sound system restarts, as the timing difference is, in general, unpredictable.

6.4 The main loop

Once the device streams are up and running, the measurement can begin. Unless recording is implemented as a separate process, the application can simply copy the contents of audio buffer to an additional measurement array, alongside the normal processing. For the measurement applications created for this thesis, the critical section is the main loop, the Karplus-Strong string model implementation. This section of the application should be optimized for low latency. Once the measurement buffer is filled, the application should stop writing.

6.5 Release of devices and program termination

After the measurement is done, the PCM devices should be properly stopped. If the system is running in shared mode, this involves unregistering the callbacks from the audio system. The measurement data must be transferred outside the application and care should be taken to avoid conversion errors. Reserved resources should be freed after securing the measurement data. Typically, the APIs provide special destructor-like functions for the application to free the memory blocks reserved via the API.

6.6 About error handling

There are numerous ways for the real-time audio program to end up being terminated abruptly, if the handling of different error situations is improperly implemented. For example, the user might unplug the USB cable between the sound card and host computer. The API itself might handle this correctly and signal the program but unless there is additional code written for this event the program might end up being deadlocked waiting the device to come back online.

Chapter 7

Latency in digital audio

In this chapter, a test application for latency measurements is presented. Additionally, test signal generation and some comparative studies about audio latency are examined.

7.1 Measuring the latency of a digital audio interface

The focus of this thesis is in the real-world performance of the actual implementations arising from the various audio application programming interfaces. One of the key properties that can be measured and compared is latency, the unwanted time between two predefined events.

For example, there is always a short delay between a key pressed by the pianist, and a sound of that key being heard. Usually, that is not called latency, as the delay is too short to be noticed, and does not annoy the artist nor the audience. Contrary to this, the time difference between a key press on a digital piano and the sound being heard from the loudspeakers, if noticeable, is called latency, as the delay in that case is considered artificial, or unnatural.

In order to gain better understanding about the results obtainable from the latency measurements, one needs to study the operations of the ideal model and evaluate the theoretical latency. In a system involving a computer and its soundcard, there generally are two separable processes. First, there is the CPU processing the data in digital format. Second, there is the sound card, or audio interface, converting the digital samples into the analog output signal, and simultaneously converting the analog input signal into a set of digital samples. Here, an assumption is made that a single audio chip is responsible for handling both the input and the output. In between these two processes, we have the mainboard bus, a communication system arrangement that transfers the samples back and forth between CPU and the audio processing unit, APU.

The latency of an audio interface of a desktop computer can be measured with ease. A short burst of white noise is written to the output device. The physical audio device output being hardwired with a sufficiently short cable to the input port of the device, the noise burst along with some noise is then acquired and written on the input buffer. The input stream is then stored on a additional buffer for further analysis, but also sent back to the output device, in order creating a loop. The structure is depicted in Fig. 7.1.

If the device amplifier gains are sufficiently adjusted, the noise burst in the loop acts similarly to a string instrument after its initial excitation. An audible ringing noise can be heard if the recording is played back through a speaker. According to Chafe et al. (2002), the perceived pitch of the sound is dependent on the total delay of the system. A typical measurement recording is shown in Fig. 7.2. As the model is run in real-time, additional steps should be taken to detect buffer underruns in the audio streams. The delay of the whole system appears as the N sample delay line in the model.

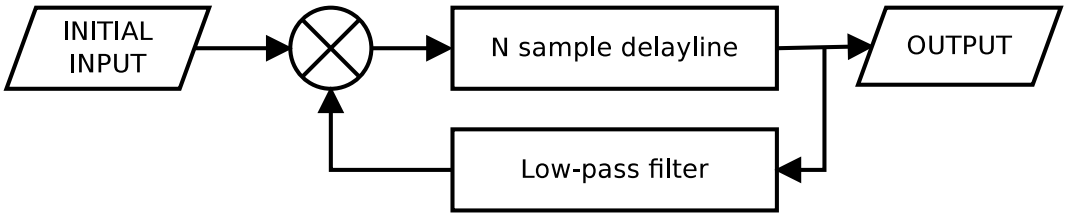

Figure 7.1: Plucked string physical model. Adapted from Karplus and Strong (1983).

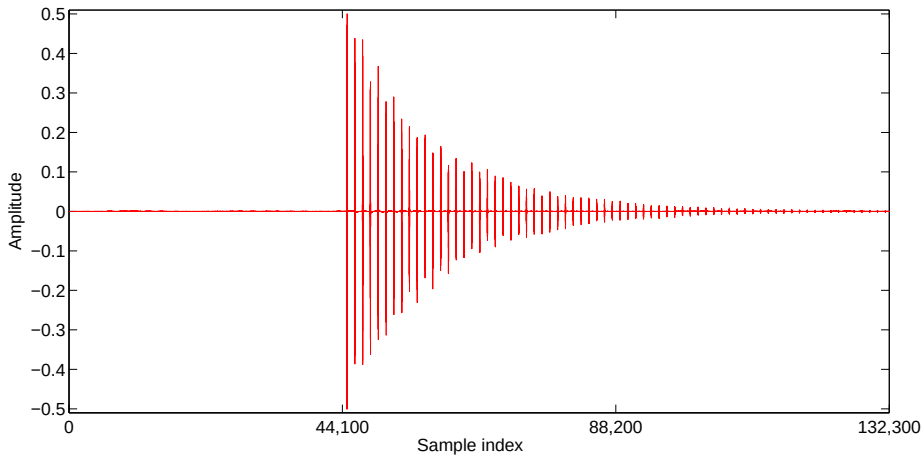

Figure 7.2: First three seconds of a typical recording of a string model application. The wavetable has been initialized at approximately one second after the recording was started. Sample rate was 44.1 kHz.

7.2 About the Karplus-Strong algorithm

For the API tests done in this thesis, a modified version of the Karplus-Strong plucked string physical model synthesis algorithm was implemented. Karplus-Strong model has been successfully used by Chafe et al. (2002) for sonification of the system delay and possible changes in it. Essentially, the wavetable of the plucked-string model is replaced by the unknown delay between the two endpoints, or in Chafe's case in particular – the Internet.

By listening to the output of this special string model in action, a musical tone can be heard if the round trip travel time lies within the range of subjects pitch sense. The resulting tones were shown to provide an intuitive evaluation of quality of service of the signal path, including latency, jitter and packet loss, so that the sonification could be used to analyze the Internet network behavior at finer granularities than what the standard ICMP ping is suited for. However, the presented technique is somewhat limited to real-time monitoring, as the signal has to be actually heard to be evaluated.

It would seem that the string model could be used in this thesis as well in order to reliably compare the different implementations. However, as shown by Liu et al. (2007), the detection of exact frequencies by hearing, even after training, is far from adequate. As in this thesis we have the possibility to record the measurement, more reliable solutions for latency measurements arise.

The algorithm, by Karplus and Strong (1983), itself was developed as an inexpensive implementation of a real-time digital synthesis of plucked-string and drum timbres. Its performance was exceptional at the time of publication, as compared to additive synthesis.

The algorithm is a modification of wavetable synthesis. The samples initially set in the buffer (of length p) are repeated over and over again in the same order, resulting in a looping sound characterized largely by the buffer length and sample rate f_{S}. The output Y at time t is

$$Y_{\mathrm{t}} = Y_{\mathrm{t}-p}. \tag{7.1}$$

The fundamental frequency f_{f} of the system is

$$f_{\mathrm{f}} = \frac{f_{\mathrm{S}}}{p}. \tag{7.2}$$

Strong's modification was to add averaging of two successive samples, which results in a formula for Y_t:

$$Y_t = \frac{1}{2}\left(Y_{t-p} + Y_{t-p-1}\right). \tag{7.3}$$

The model is depicted in Fig. 7.3. Due to the two-point moving-average filtering, the higher harmonics of the sound decay more rapidly than the lower ones. It happens to produce a decay of the waveform that sounds like the decay of a plucked string. Independent of the initial spectrum the sound decays to a sine wave and furthermore to a constant value. In order to produce realistic string sound, the wavetable should be filled with random values although any desired waveform can be used, as in pure wavetable synthesis.

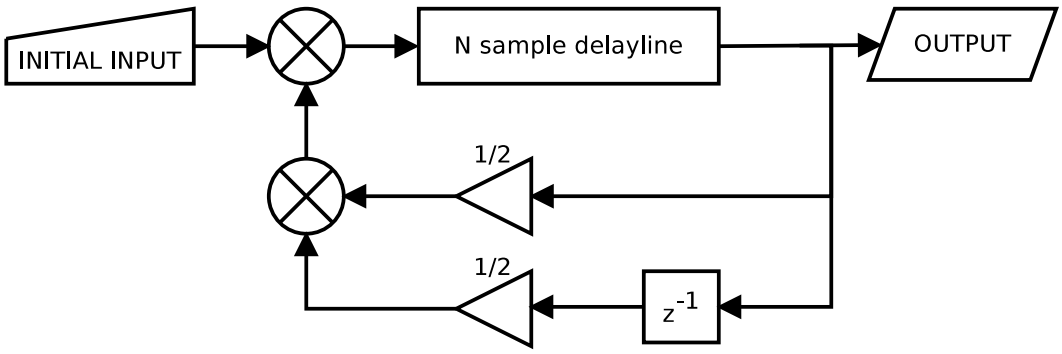

Figure 7.3: Plucked string physical model with lowpass filter visible. Adapted from Karplus and Strong (1983).

7.3 About the physical measurement setup

A large number of real-world, real-time measurements were made for this thesis. The APIs are for all-purpose operating systems. An integrated sound card of a laptop computer was chosen as the measurement system. The setup can be assumed to be susceptible to noise. The test signal was selected as a finite length, digital, pseudo-random white noise burst. A near maximal (0.95) amplitude was used to prevent possible side-effects of clipping, and still maintain an adequate signal-to-noise ratio. Due to the nature of the pseudorandom test signals selected, the system to be measured does not have to be particularly linear or time independent.

A degree of causality of the system is assumed. When interpreting measurement recordings, the noise burst detected first is assumed to be sent first. Consecutive echos, as depicted in Fig. 7.4 are assumed to be in order as well. As PC operating

systems today are multitasking and capable of genuinely parallel processing, this assumption can occasionally be outright wrong. Moreover, the measurement setup can be seen consisting of mostly black boxes, of which precise functions are unknown. There is no obvious correlation between a line of C source code and the resulting set of processor instructions after compilation and linking this code with a variety of compilers of different vendors, versions, and configurations.

The delay can be easily determined graphically as the distance between the noise "echos" in the time domain presentation, as long as the initial noise burst was shorter than the delay. For example, in Fig. 7.4, the first two "echos" are shown along with the initial excitation noise burst. Based on the loop structure, latency in the system is the difference between consecutive noise bursts measured from a stationary point along the Karplus-Strong implementation. The measurement point can be chosen freely.

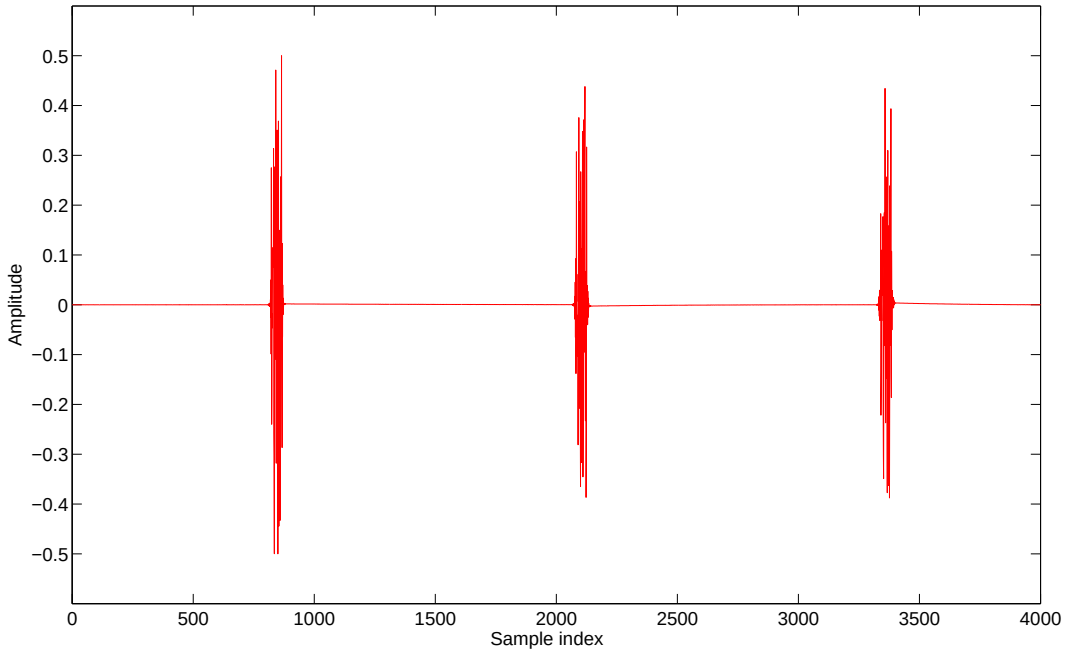

Figure 7.4: Karplus-Strong measurement data consisting of original excitation signal and two sequences of loopback. The latency can easily be determined by counting the number of samples between consecutive noise bursts.

7.4 Test signal generation

For the Karplus-Strong algorithm to work, its wavetable must be seeded with something. As an analog, a string of a violin must be initially excited by some force to enable vibrations thus to produce the distinctive sound. For the signal used in the Karplus-Strong model for this thesis, the excitation signal is pseudorandom white

noise.

A pseudorandom number generator used in computers is seeded with some initial number. Often some random variable is selected, for example by reading some uninitialized memory locations and combining the result. The random number generator of the standard C library, `rand()`, is initialized by the command `srand(31337)`. As stated by the srand command manuals, the sequences of `rand()` are repeatable by calling `srand()` with the same seed value, which is what was done in the measurement programs written for this thesis. It should be noted that the output of `rand()` is an integer in the range from 0 to `RAND_MAX`, the latter being a compile time constant.

A total of 64 integer values can now be extracted. They are not suitable for use as is, as the audio signal is usually scaled from -1.0 to +1.0, so the integer values have to be normalized using the compile-time constant `RAND_MAX`. Additionally, the signal should be a zero-mean sequence. The final signal should also be scaled to maximize the input level while preventing it from clipping. Finally, the normalized floating-point version of the test signal was generated using the C source code snippet as shown in listing 7.1. Same logic (without the printing to standard output) was used in the measurement applications created for this thesis.

Listing 7.1: C99 program that prints normalized pseudorandom floating-point numbers

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5    srand(31337);
6    for ( int i=0; i < 64; i++ ) {
7      float normalizedRnd = (float) ( rand() / (1.0f * RAND_MAX) - 0.5f );
8      printf("%2u: %f\n", i+1, normalizedRnd);
9    }
10   return EXIT_SUCCESS;
11 }
```

64 floating point numbers, with their values normalized, can then be extracted. The signal could be further amplified by multiplying with ratio $\frac{0.99}{0.5}$. The test signal is then sent to the Karplus-Strong model where it will be constantly filtered with the two-point moving-average filter.

7.5 Comparable studies about the latency of desktop audio

As digital audio signal processing is increasingly done on common desktop operating systems, it has become reasonable to measure and study the latency of those systems.

The previous work of Wang et al. (2010) is partially comparable to this thesis as the reasoning behind platform choices are similar. The team decided to use Intel based Apple computers as the test platform as the main test platforms as they supported all three popular operating systems: Mac OS X, Linux and Windows. The author (of this thesis) would like to note that this situation is due to the restrictive licensing of Apple's operating system and not a strictly technical issue.

The measurement techniques in the study were quite different from the one used in this thesis. In their setup, the test signals were split into two channels, of which one was sent through the operating system and the second directly to the multichannel recorder. Latency was then observed by measuring the time difference between the two channels. Wang et al. also tested the effect of different processor loads on the audio latency of these modern multitasking operating systems, and found that in low latency modes of operation the audio signals would suffer losses and distortion if the host CPU was stressed by calculations within the audio application. Interestingly, the study finds that the CPU stress by external applications had unnoticeable effect.

Wang et al. compared DirectSound, MME, Core Audio and ALSA. For some reason, they omitted the more low-level WASAPI from their Windows tests. Many commercial software products allow the users to 'measure' the latency of their current system. Important observation in the work was that the reported latencies of software audio hosts (like Logic and Ableton) did not match the values given by actual measurements.

Much earlier studies by MacMillan et al. (2001) also compared Windows, Mac OS and Linux, but they only used a single sample rate of 44.1 kHz. The studies were more extensive though, as multiple versions of each operating system was tested, and also multiple APIs on Windows (Steinberg ASIO, DirectSound and MME), Mac OS (Apple SoundManager, Core Audio) and Linux (OSS and ALSA). As the team could not perform all the tests on the same hardware and they mixed results from professional soundcards with ones taken from consumer grade cards, the comparability of those measurements between platforms can easily be disputed. The team concluded that reliable low-latency performance could not be expected from the desktop operating systems of that time.

Chapter 8

Measurements

In this chapter, the practical aspects of latency measurements are discussed, and the software and hardware details of the measurement platform are examined. In order to compare complexity, the number of methods exposed by the APIs are compared. Additionally, the results of the latency measurements are given, along with some statistical analysis and discussion.

8.1 Latency measurement in practice

In the measurements done for this thesis, a chunk of audio data is first written to the sound card output buffer. Sound card electronics then transfer the digital stream to hardware DAC, where it is eventually converted to an analog signal. This is then fed via a wire to another audio device to be converted back to a digital signal and eventually written to its audio input buffer. As the process is partially asynchronous, there is arbitrary amount of buffering in the host software, host hardware and possible audio device software in addition to the audio device hardware limitations. The resulting round trip latency typically varies between audio system restarts.

In the measurements done for this thesis, the sample format conversions have been deliberately avoided. Instead, all the devices are mandated to use identical sample formatting, thus eliminating the need for cumbersome sample format conversions. All the components are set to operate at identical sample rates, again to eliminate the need for upsampling or downsampling. This should reduce the possible points of failure in the measurement process. In theory, this approach should also greatly improve the reliability of the results, as the latencies are not affected by potentially poor conversion algorithms.

Nevertheless, actual results may still vary depending on the hardware implementations being used. All measurements are made while running the sound card on a two-channel “stereo” mode. In shared mode operations, where the measurement application shares the audio buffers with other user space components, the operating

system chooses the proper buffer size. Additionally, ALSA has software devices that silently convert between rates and formats.

8.2 About sample formats

For various historical reasons, a large number of different sample rates are being used. The most prominent is the CD-DA sample rate of 44100 Hertz. Most of the professional digital audio gear operates at 48 kHz or at its multiple. The need to convert data from a sample rate to another often arises in practice.

For an uncompressed PCM stream, there are two kinds of sample rate conversions. In upsampling, the sample rate of the target stream is increased from the sample rate of the original stream. In downsampling, the sample rate is decreased. Downsampling generally implies loss of data, as some of the samples in the original stream cannot be copied to the new stream with less room for samples. Upsampling may or may not lose information, depending on the material and rates being used.

Sample format conversions can be thought as an umbrella term consisting of not only sample rate changes, but also any kind of channel mixing, compression or uncompression and bit depth or endianness changes. For plain PCM, common operations are conversion from a floating point presentation to a fixed-point or from floating point to signed or unsigned integer formats.

The sample endianness conversion is an important part of device compatibility. Basically there are two types of endianness: big and small. Big-endian systems store and transfer the most important part of values first. This might be seen as a natural way as many human writing systems are based on writing numbers from left to right so that the `most significant` decimals are written first (on the left). This notation makes it easier to immediately see the approximate size of the value.

In order to execute elementary calculations, like multiplication and addition, we need to handle the least significant parts first. As computers are all about calculations, the natural presentation for them is small-endian, where the `least significant` binary digits are transferred first.

In this thesis work, sample rate conversions are deliberately eliminated. This firstly simplifies the code, reducing the possible points of failure in the measurement process. Secondly, this greatly improves the reliability of the results, as the latencies are not affected by poor conversion algorithms.

One special case is the presentation of multiple samples. Even 24-bit samples (regardless of their interpretation) can be stored using different techniques. For computers,

Table 8.1: Storage of 24-bit sample in a 32-bit register

$r_0r_1...r_6r_7$	$r_8r_9...r_{14}r_{15}$	$r_{16}r_{17}...r_{22}r_{23}$	$r_{24}r_{25}...r_{30}r_{31}$
$b_0b_1...b_6b_7$	$b_8b_9...b_{14}b_{15}$	$b_{16}b_{17}...b_{22}b_{23}$	`unused but reserved`

storage of words consisting of 24 bits is tricky, because the registers and buses generally expect the data to appear as powers of two. Generally, while processing the 24-bit samples, the computer treats them as 32-bit ones. Additional zero bits are used to pad the sample to 32-bit presentation. This has the obvious disadvantage of wasting exactly 25 percent of storage and transmission bandwidth. To overcome this, some soundcards employ a special 3-byte presentation. As illustrated in Table 8.1, several 24-bit samples are chained together to be stored as a series of 32-bit samples. This chaining requires compression and decompression phases.

In ADC, the sampler performs a conversion of a continuous-valued signal into a discrete-valued sequence of digital samples. These samples have a discrete value whose precision depends on the bit depth of the sampler. For the audio end use, the "CD quality" bit rate is as low as 16 bits per sample. With 16 bits of depth, the headroom between the maximum and the theoretical minimum sound pressure levels is 96 dB. To fully utilize this headroom in a fixed-point presentation, the signal gain must be set appropriately to let the actual signal values of the material span across the available headroom.

The canonical audio data format `AudioSampleType` of the Core Audio framework is a typedef for `Float32` or `SInt16`. The first is a linear PCM format using 32-bit floating-point numbers with one sign bit, eight exponent bits, and 23 fraction bits. The second is linear PCM format using 16-bit integers. The format depends on the `CA_PREFER_FIXED_POINT` precompiler macro, defined in `CoreAudioTypes.h` header file. On Mac OS X, the floating-point version is used. `SInt16` is only used if the target platform for the application is "iPhone OS". According to Apple Inc. (2007), this is to get faster calculations and less battery drain on the mobile hardware. Apple recommends using their canonical type `AudioSampleType` as is, instead of using `Float32` or `SInt16` directly.

WASAPI, as well as other Windows Core Audio methods, leave the programmer with some choice considering the sampling format for audio streams. In exclusive mode the programmer can try to initialize the audio client with whatever format the device supports. For a non-exclusive or a shared mode, the application should query the format used by the audio engine via the `IAudioClient::GetMixFormat` method. The format returned may or may not be directly supported by the actual audio endpoint device. If not, the audio engine will handle the appropriate conversions (from floating point to integer values, for example) on run-time. (Microsoft Corporation, 2012a)

With ALSA, the programmer is responsible for handling the proper audio stream format in the application. The mode of the device must correspond to the C type used

in the buffers of the application, as there is no automatic conversion by default. API sample formats and corresponding C types used in this thesis are listed on Table 8.2.

Table 8.2: Comparison of ALSA, Core Audio, WASAPI, and corresponding C data types

API	API data type	C data types
ALSA	SND_PCM_FORMAT_FLOAT64	Float64, double
ALSA	SND_PCM_FORMAT_FLOAT	Float32, float
ALSA	SND_PCM_FORMAT_S16	SInt16, signed short
Core Audio	AudioSampleType	Float32, float
Core Audio	AudioSampleType	SInt32, signed int
Core Audio	AudioSampleType	SInt16, signed short
WASAPI	KSDATAFORMAT_SUBTYPE_IEEE_FLOAT	float

8.3 About the measurement platform

The measurements were done on an Apple MacBook Pro laptop shown in Fig. 8.1 using its internal sound card. Headphone output was hardwired to the line input using a short 1.5-meter stereo cable.

The author was not able to find the exact chip details of the Late 2008 MacBook Pro (model number A1261) unit. On Windows OS, the chip shows up as ALC885 by Realtek Semiconductor Corporation. On Linux 3.2.0 kernel however, the `lspci` tool identifies this audio device as `82801H (ICH8 Family)`, whereas through the ALSA API the individual devices show up as `ALC889A`. Additionally, both Wang et al. (2010) and Dang (2005) have previously determined the audio devices of the same MacBook Pro series to be ALC885.

As stated by both the workstation manufacturer Apple and the likely manufacturer of the audio chip of the workstation, Realtek, the audio system conforms to the High Definition Audio Specification by Intel Corporation (2010). The specifications reveal some details about the behavior of the system. Any systems conforming to the HD Audio specifications are *isochronous*. They provide their own timing from a 25 MHz clock source. In addition, HD Audio devices cannot be synchronized with external sources, only exception being the S/PDIF input stream, if available. For example, Microsoft has gone as far as to made the 128-byte alignment (required in the HD Audio) mandatory in WASAPI, so that audio devices cannot be initialized using buffer lengths of different alignment, even if the device itself would accept them. (Microsoft Corporation, 2013)

Realtek ALC chip family details are published by Realtek Semiconductor Corporation (2013). The datasheet lacks information about the delay caused by the converters.

Figure 8.1: The measurement setup. 1.5 m stereo cable is used to connect headphone output of the laptop to its line input.

Wang (2011) has studied the latency of the HD Audio codec. Based on his measurements, a signal travelling through the converters will be delayed 1.2 ms at 44.1 kHz, 1.1 ms at 48 kHz, and 0.6 ms at 96 kHz.

In order to create the measurement applications, a variety of interconnected software components were used. It is understandable that in complex software there are quite a lot of errors. To enable the reconstruction of the measurements and aid in the possible identification of the flaws that affected the results, the details of the measurement system operating system, compiler, and programming language version is detailed here.

Core Audio was tested using Apple Mac OS X 10.7.5 Lion with Darwin kernel version 11.4.2. The measurement program was compiled with Apple Xcode version 4.5.1.

ALSA was tested using Canonical Ubuntu 12.04.2 LTS (Precise Pangolin) with x86_64 series Linux kernel version 3.2.0-44-generic. The measurement application was compiled with Free Software Foundation Inc. GCC (Ubuntu/Linaro 4.6.3-1ubuntu5) version 4.6.3. No special kernel optimizations were being used.

WASAPI tests were run on Microsoft Windows Server 2008 R2 64-bit Service Pack

1 that had the Windows Audio service enabled. The application was compiled with Microsoft Visual Studio 2010 Professional Service Pack 1.

In all cases, the measurement software was being run in user space (without privileged access or elevated credentials). Access of the application to the audio device was exclusive (where available). No other audio applications were deliberately run simultaneously.

Source code for the measurement applications was written in C99 where available, and in Visual C++ 2010 where not.

8.4 Counting the API methods

As described in Chapter 5, in order to get some comparability for the usability, the methods exposed by the APIs are counted.

In order to utilize WASAPI methods, the audio program has to include `audiopolicy.h`, `Audioclient.h` and `MMdeviceapi.h`. `Audioclient.h` in turn includes ten header files, `windows.h` among them. Many of these are general system headers and thus should not be counted towards the API payload. `AudioSessionTypes.h` is one that should be, but in it, there are only some internal magic number definitions and no public methods.

In `Audioclient.h`, Microsoft declares total of nine classes, `IAudioRenderClient` and `IAudioCaptureClient` among them. Calculating the methods within those classes, it was determined that the total payload of `Audioclient.h` was 36 methods. In addition, there are 33 methods from `audiopolicy.h`, and 18 more from `MMdeviceapi.h`, totalling 87 methods.

In order to utilize ALSA methods, the audio program has to include one header file, the `asoundlib.h`. This in turn includes numerous other header files: `conf.h` declares 56 methods that are only used internally by the API, thus useless to the application programmer. `control.h` declares a total of 253 methods, whereas `error.h` declares only 6 methods and `global.h` 20 methods, `hwdep.h` 45, `input.h` 8, `mixer.h` 119, `output.h` 10, `pcm.h` 358, `rawmidi.h` 55, `seq.h` 280, `seq_midi_event.h` 10, `seqmid.h` 49 and `timer.h` 90. There were no methods declared in `seq_event.h`. A total number of methods exposed in ALSA headers is 1359.

In order to utilize Core Audio methods, the audio program has to include one header file, the `CoreAudio.h`. This in turn includes three other header files: `CoreAudioTypes.h`, `AudioHardware.h` and `HostTime.h`. Following through these, more headers get included: `AudioDriverPlugIn.h`, `AudioHardwareBase.h`, `AudioHardwareDeprecated.h` and

finally `AudioHardwarePlugIn.h`. A total of 74 functions are declared in these files.

To compare the complexity of the three APIs, the number of methods was counted for each API, and are given in Table 8.3.

Table 8.3: Methods exposed by APIs

API	Number of methods
Core Audio	74
WASAPI	87
ALSA	1359

8.5 Measured latencies

In this thesis, a number of audio programming interfaces were studied by implementing a Karplus-Strong string model, running it on the same hardware platform while varying parameters, and measuring the resulting round-trip latencies. A list of measured raw latencies is given in Table 8.4.

The sample rates 44.1 kHz, 48.0 kHz and 96.0 kHz were chosen as those to match the most common ones known to be used in the audio industry, that were also supported by the measurement hardware at hand. At first, 256-frame buffer size was intended to be used as the basis of measurement series. Later on, it was found that the measurement device could not be started with this buffer length in 96 kHz WASAPI exclusive mode, so the next available length of 288 frames was selected instead. To analyze the effect of changes in the buffer length, other measurement points were selected as the double and the quadruple of 288 frames, resulting in the series 288, 576 and 1152.

Measured latencies are illustrated in Fig. 8.2. Mainly due to the linear scale utilized in the figure along with exponentially spaced measurement points for buffer sizes, the group of measurements done both with the highest buffer length, and low frame rates, stands out. Latencies in this group are roughly twice the latencies taken from the corresponding measurements that were done with the buffer size halved.

An interesting anomaly can be seen by observing the 96 kHz WASAPI results. Those latencies are much lower than corresponding ALSA or Core Audio results at equal measurement points. Closer inspection reveals the difference to be approximately one buffer length. As stated earlier in Section 8.1, this is likely not an error, but merely a phenomenon resulting from asynchronous nature of the complex system. Running at 96 kHz sample rate, the WASAPI-based application seems to have initialized the input and output devices consistently in synchronization, exhibiting an optimum delay of one buffer size instead of two, whereas this behavior did not occur

while at other sample rates.

In order to compare APIs against each other, the abovementioned latencies L_{RAW} are compensated for the naturally occurring delay resulting from the buffer size B and sample rate f_{S} used at each point:

$$L_{\mathrm{OVERHEAD}} = L_{\mathrm{RAW}} - \frac{B}{f_{\mathrm{S}}}. \tag{8.1}$$

What remains is the overhead latency L_{OVERHEAD}, illustrated in Fig. 8.3. Here, with three exceptions apparently arising from the abovementioned WASAPI anomaly, Core Audio stands out with all the highest latencies, whereas ALSA application consistently exhibits the lowest latencies. All the measured values are within a 3 ms margin.

8.6 Statistical analysis

Due to the relatively low number (15) of measurements per measurement point, the uncertainties of the latency values given in Table 8.4 are calculated using Student's t-distribution instead of normal distribution. In other words, population standard deviation being unknown, the confidence interval for the population mean must be adjusted when using sample deviation instead of population standard deviation. First, the unbiased sample variance of the data in a measurement point is calculated as given in Eq. 8.2.

$$s_{N-1}^2 = \frac{1}{N-1} \sum_{j=1}^{N} \left(x_j - \frac{1}{N} \sum_{i=1}^{N} x_i \right)^2 \tag{8.2}$$

Selecting 95 percent confidence interval, where the number of measurements N per measurement point is 15 and therefore the degree of freedom f being 14, the Student's t-distribution adjustment value Z is 2.145. As shown in Eq. 8.3, at 95 percent confidence, true mean at the measurement point, based on 15 measurements taken at that the point, lies between the given values. The population mean $\mu_{95\%}$ is thus *probably* between the two values.

$$\Pr\left(\overline{x} - \frac{Z s_{N-1}}{\sqrt{N}} \leq \mu_{95\%} \leq \overline{x} + \frac{Z s_{N-1}}{\sqrt{N}} \right) = 95\% \tag{8.3}$$

Table 8.4: Summary of latency measurements (with 95% confidence level, the population means of measured latencies lie in the confidence intervals given in the latency column) of a Karplus-Strong model built on different APIs

API & mode	Sample rate	Buffer size	Latency / ms
WASAPI exclusive	96 kHz	288	5.2403 ± 0.8
ALSA HW mode	96 kHz	288	6.8653 ± 0.01
WASAPI exclusive	96 kHz	576	7.0549 ± 0.01
Core Audio Aggregate	96 kHz	288	8.7743 ± 0.6
ALSA HW mode	96 kHz	576	12.865 ± 0.01
ALSA HW mode	48 kHz	288	13.082 ± 0.07
WASAPI exclusive	96 kHz	1152	13.117 ± 0.06
WASAPI exclusive	48 kHz	288	13.413 ± 0.4
Core Audio Aggregate	96 kHz	576	13.733 ± 0.02
ALSA HW mode	44.1 kHz	288	14.192 ± 0.06
WASAPI exclusive	44.1 kHz	288	14.455 ± 0.1
Core Audio Aggregate	48 kHz	288	15.411 ± 0.01
Core Audio Aggregate	44.1 kHz	288	16.772 ± 0.03
ALSA HW mode	96 kHz	1152	24.842 ± 0.05
ALSA HW mode	48 kHz	576	25.078 ± 0.08
WASAPI exclusive	48 kHz	576	25.311 ± 0.05
Core Audio Aggregate	96 kHz	1152	25.725 ± 0.02
ALSA HW mode	44.1 kHz	576	27.327 ± 0.05
Core Audio Aggregate	48 kHz	576	27.436 ± 0.02
WASAPI exclusive	44.1 kHz	576	27.548 ± 0.05
Core Audio Aggregate	44.1 kHz	576	29.820 ± 0.02
ALSA HW mode	48 kHz	1152	49.119 ± 0.04
WASAPI exclusive	48 kHz	1152	49.390 ± 0.05
Core Audio Aggregate	48 kHz	1152	51.422 ± 0.03
ALSA HW mode	44.1 kHz	1152	53.415 ± 0.05
WASAPI exclusive	44.1 kHz	1152	53.665 ± 0.08
Core Audio Aggregate	44.1 kHz	1152	55.938 ± 0.03

8.7 Aiming for low latency

In this thesis, three low-level audio application programming interfaces were tested, one for each major desktop operating system, ALSA for Linux, Core Audio for Mac OS X, and WASAPI for Windows. The selected APIs were studied by implementing a real-time audio test program with each API, and by running the test programs on the same hardware.

Putting API differences aside for a while, and looking at the results, some conclusion can be made. With a few exceptions, measurements done using the largest buffer size had the highest total latency. Accordingly, measurements done using a low sample rate resulted in a high total latency. Moreover, the lowest latencies were observed

while using both high sample rate and short buffer length. There are, obviously, limits in software and hardware on how much these can be optimized. The buffer sizes used in the measurements in this thesis were selected conservatively. The hardware was found to be capable of operating with even tighter settings than was used.

The purpose of this thesis, however, was to compare APIs against each other. In Fig. 8.3 the results are compensated for the "known" latency resulting from buffer size and sample rate. What remains is the overhead latency. Overhead latency is part of the latency that cannot be explained by differences in the measurement hardware (as it was kept the same) or by saying it is a natural part of the process (as the effect of buffer size was eliminated). Comparing the difference of overhead latencies between APIs gives an insight about their efficiency.

This approach obviously still has some drawbacks. Although the hardware is the same, variables ranging from firmware to operating system kernel priorities cannot be sufficiently eliminated. Although we certainly can compare the latencies between different approaches, the reader is advised not to make any far-fetched conclusions.

Compared to the other two APIs that were studied, ALSA has strikingly large number of methods. Despite ALSA being the most cumbersome application programming interface, the test application utilizing the sound system via this framework has the lowest latencies. Accordingly, the tradeoff for selecting the easiest-to-use API, Core Audio, is the clearly highest overhead latency.

Interestingly enough, the best performance of the given Apple hardware was seemingly only obtainable by installing a non-Apple operating system. WASAPI exhibited good performance despite being well-documented and easy to use.

8.8 Making the decisions count

During the development of measurement applications for this thesis, it was observed that by default WASAPI and ALSA do not let the application programmer to make decisions. ALSA software devices even give a false impression that buffer size and sample rate could be changed, whereas they silently use conversion behind the scenes. Therefore it is imperative to carefully check that the audio card is being operated using its ALSA hardware device implementation. Due to convention, ALSA device names beginning with `hw:` are pure hardware devices without any conversions. For WASAPI, the exclusive mode must be selected in order to change the buffer size.

Additional challenges arise from the usage of non-synchronizable audio hardware. As a protection against interruptions in audio stream, a variety of buffer implementations are used along the signal path. The obvious tradeoff is the increased latency of the sound system. It was found that the multiple asynchronous buffers on the audio

pipeline result in arbitrary total latency that varies between sound system restarts, even if the hardware setup is kept unchanged.

8.9 Comparison of API documentation and IDEs

The major drawback of ALSA is the lack of professionally written documentation for application programmers. Audio application programmers willing to adopt basic API functionality need tutorials with some sample code examples along with the API library documentation. In ALSA documentation, the latter is extensively provided while the former is outdated or missing. Following the examples given by the Microsoft Developer Network Library and Mac Developer Library, it was fairly trivial to write a working real-time application with simultaneous input and output, even without previous experience in neither of the platforms.

For some reason, exact instructions on how to implement simultaneous input and output had been omitted from all three API documentations. Luckily, it was possible to combine the separately given examples for capture and playback.

For Windows and Mac OS X platforms, there are de facto graphical integrated development environments, Visual Studio and Xcode, which are specifically built to assist the programmer. They bring the API documentation into action by helping the programmer in selecting the appropriate methods and in filling in the correct system call parameters. Essentially, they provide a text-editor with source code color-coding capabilities in order to assist the programmer to avoid syntax errors.

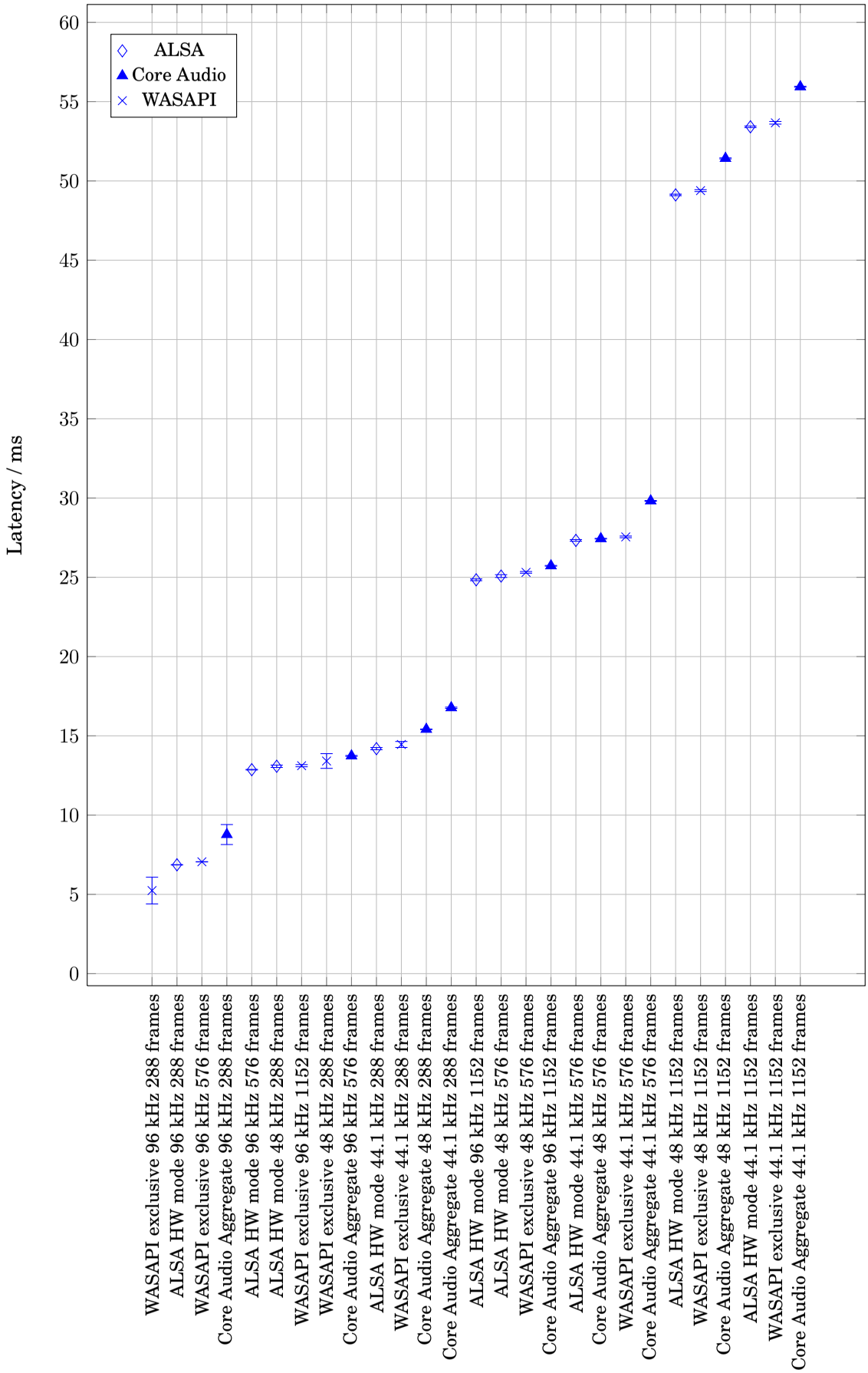

Figure 8.2: Total latencies of the Karplus-Strong string model build on different APIs and run with a variety of buffer lengths and sample rates.

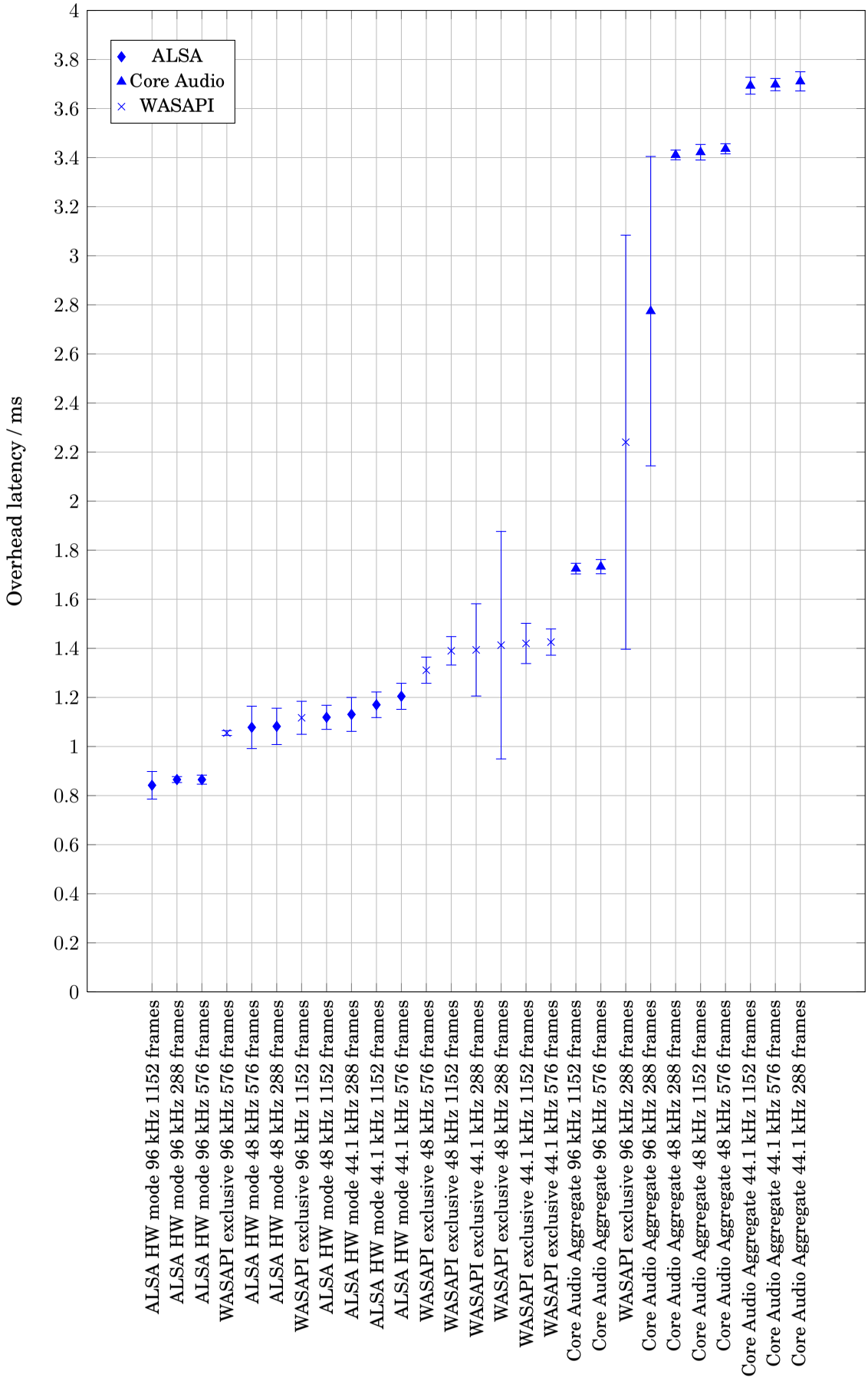

Figure 8.3: Overhead latencies of the Karplus-Strong string model build on different APIs and run with a variety of buffer lengths and sample rates.

Chapter 9

Conclusion and future work

In the field of digital audio processing, a need to rapidly prototype a measurement or signal generator program often arises. Therefore, for an audio application programming interface to be successful, the required time to master the methods necessary to build a real-time, simultaneous input and output application should be reasonably short. For low latency, it is expected to be feasible to use the lowest level of abstraction obtainable from the APIs. As the higher level interfaces are layers on top of the lower ones, the increased easiness of use and safety of operations are, in fact, traded for higher latency.

Of the numerous audio application programming interfaces available for personal workstations today, three low-level ones were chosen to be tested, one for each major desktop operating system, ALSA for Linux, WASAPI for Windows, and Core Audio for Mac OS X. The selected audio application programming interfaces were studied and their real-world, real-time performance was compared. A modified Karplus-Strong string model patch, that had its wavetable replaced with the unknown delay between the endpoints, was implemented with each API and was run on the same hardware. The results generally supported the well-known idea that low latency can be achieved by selecting a high sample rate and short buffers, where available. Considering that the differences between measured overhead latencies were within three milliseconds, choosing the optimal settings for any given API is expected to yield more than adequate performance improvement, whereas actually changing an API mid-project should be seen as the last resort.

Based on the measured latencies, the total latency of the system could be decreased by increasing the sample rate and decreasing the buffer size. Apart from this, the latency of the system was found not to stay constant across software implementations, as all tests were run on the same hardware. By compensating the effect of buffer size and sample rate, the overhead latency characteristic of each implementation was extracted from the results. All overhead latencies were found to be within a few milliseconds.

The smallest overhead latencies were measured from the ALSA implementation at 96 kHz. Overall, ALSA gave the best performance, and WASAPI was nearly as good. The largest overhead latencies were measured from the Core Audio implementation both at 44.1 kHz and 48 kHz sample rates. Conversely, Core Audio had the least number of methods exposed to the programmer. ALSA, on the other hand, was found to have a considerably larger number of methods exposed to the programmer. However, the finding that ALSA also offered the lowest overhead latency compensates this drawback. WASAPI performed well in both fields, by exposing only a few more methods than Core Audio, and by having nearly as low latencies as ALSA.

Summing up, all three APIs can be used to create a working, real-time, audio application. Those with the luxury can pick any and succeed. The differences in I/O latency observed in this thesis might have changed as having been addressed in later software updates to the operating systems, APIs, or device drivers. In terms of reducing latency in applications, it definitely pays to properly learn the APIs one is using. Real-time audio programming remains difficult and challenging, but also very doable and rewarding.

Only a narrow selection of APIs was studied in this thesis. For future work, the author recommends a full spectrum study including APIs that were left from this one, namely at least ASIO, DirectX Audio, JACK, OpenAL, PortAudio, PulseAudio, and SDL. While true that these run on top of the lower level APIs, it should be interesting enough to measure how close they can get in terms of latency. If the API is sufficiently easy to use or offers any unique advantages, it might be reasonable to relax on the latency requirements slightly. In addition to PC hardware, also mobile devices, iOS, Android, and Windows mobile should be studied. Tests on typical multichannel setups could be covered as well.

Bibliography

Larry C. Andrews and Ronald L. Phillips. *Mathematical Techniques for Engineers and Scientists*. SPIE, 2003. ISBN 9780819478290 (electronic). URL http://dx.doi.org/10.1117/3.467443.

James Angus. *Direct DSP on Sigma-Delta Encoded Audio Signals*. In Audio Engineering Society Conference: UK 14th Conference: Audio - The Second Century, Jun 1999. URL http://www.aes.org/e-lib/browse.cfm?elib=7979.

James Angus. *Achieving Effective Dither in Delta-Sigma Modulation Systems*. In Audio Engineering Society Convention 110, May 2001. URL http://www.aes.org/e-lib/browse.cfm?elib=10026. Last retrieved in the 27th of October, 2013.

Apple Inc. *Core Audio Overload Warnings*. 2006. URL https://developer.apple.com/library/mac/qa/qa1467/_index.html. Last retrieved in the 28th of September, 2013.

Apple Inc. *Core Audio Overview*. 2007. URL http://developer.apple.com/library/mac/documentation/MusicAudio/Conceptual/CoreAudioOverview/CoreAudioOverview.pdf. Last retrieved in the 27th of October, 2013.

Apple Inc. *Apple Open Source Releases*. 2013. URL http://opensource.apple.com/. Last accessed in the 10th of January, 2013.

Attila Baldini. *Digital Mixing Console Based on a Single Low-Cost DSP*. 1997. URL http://www.aes.org/e-lib/browse.cfm?elib=7291. Audio Engineering Society Convention 102.

Ross Bencina. *Real-time audio programming 101: time waits for nothing*. Aug 2011. URL http://www.rossbencina.com/. Last retrieved in the 28th of September, 2013.

Chris Chafe, Scott Wilson, and Daniel Walling. *Physical model synthesis with application to Internet acoustics*. In Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on, volume 4, pages IV–4056–IV–4059, May 2002. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5745548&isnumber=5745339. DOI 10.1109/ICASSP.2002.5745548.

Alan Dang. *Apple MacBook Review: Part 1*, Jan 2005. URL http://www.tomshardware.com/reviews/apple-macbook-laptop,2130-2.html. Last accessed in the 25th of September, 2013.

Paul Davis. *A Tutorial on Using the ALSA Audio API*. 2002. URL http://equalarea.com/paul/alsa-audio.html. Last accessed in the 10th of January, 2013.

Paul Davis. *What is JACK?* 2011. URL http://jackaudio.org/. Last retrieved in the 24th of November, 2013.

U. Dekel and J. D. Herbsleb. *Improving API documentation usability with knowledge pushing*. 2009. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5070532&isnumber=5070493. IEEE 31st International Conference on Software Engineering. DOI 10.1109/ICSE.2009.5070532.

E. Maurice Deloraine and Alec H. Reeves. *The 25th anniversary of pulse code modulation*. IEEE Spectrum, 1965. DOI 10.1109/MSPEC.1965.5212943.

Chris Dunn and Mark Sandler. *Psychoacoustically Optimal Sigma-Delta Modulation*. J. Audio Eng. Soc, 45(4):212–223, 1997. URL http://www.aes.org/e-lib/browse.cfm?elib=7862.

Edward F. Evans. *Basic Physiology of the Hearing Mechanism*. In Audio Engineering Society Conference: 12th International Conference: The Perception of Reproduced Sound, Jun 1993. URL http://www.aes.org/e-lib/browse.cfm?elib=6253.

Harvey Fletcher and W. A. Munson. *Loudness, Its Definition, Measurement and Calculation*. The Journal of the Acoustical Society of America, 5(2):82–108, 1933. URL http://link.aip.org/link/?JAS/5/82/1. DOI 10.1121/1.1915637.

Christer Grewin. *A Format for Contribution of Digital Studio Quality Sound Signals*. In Audio Engineering Society Conference: UK 3rd Conference: AES/EBU Interface, Sep 1989. URL http://www.aes.org/e-lib/browse.cfm?elib=5341.

Edward W. Herold. *Audio for the Elderly*. J. Audio Eng. Soc, 36(10):798–800, 1988. URL http://www.aes.org/e-lib/browse.cfm?elib=5127.

Yoshimutsu Hirata. *Digitalization of Conventional Analog Filters for Recording Use*. J. Audio Eng. Soc, 29(5):333–337, 1981. URL http://www.aes.org/e-lib/browse.cfm?elib=3909.

Tomlinson Holman and Frank S. Kapman. *Loudness Compensation: Use and Abuse*. J. Audio Eng. Soc, 26(7/8):526–536, 1978. URL http://www.aes.org/e-lib/browse.cfm?elib=3261.

Jr. Hutchins, Bernard A., and Walter H. Ku. *Audio Frequency Applications of Integrated Circuit Analog Delay Lines*. In Audio Engineering Society Convention 51, May 1975. URL http://www.aes.org/e-lib/browse.cfm?elib=2400.

Intel Corporation. *High Definition Audio Specification. Revision 1.0a.* 2010. URL http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf.

Yukio Kagawa, Noboru Kyouno, Tsuyoshi Usagawa, and Tatsuo Yamabuchi. Acoustic response simulation of a cone-type loudspeaker by the finite element method. In *Audio Engineering Society Convention 117*, Oct 2004. URL http://www.aes.org/e-lib/browse.cfm?elib=12899.

Kevin Karplus and Alex Strong. *Digital Synthesis of Plucked-String and Drum Timbres*. Computer Music Journal, 7(2):43–55, 1983. URL http://www.jstor.org/stable/3680062.

Walt Kester and Analog Devices Inc. *Data conversion handbook*, 2005. URL http://www.analog.com/library/analogdialogue/archives/39-06/data_conversion_handbook.html.

Daigo Kuniyoshi, Hajime Ohtani, Junichi Okamura, Ryouta Suzuki, Kenzo Tsuihiji, and Akira Yasuda. *A Novel Universal-Serial-Bus-Powered Digitally Driven Loudspeaker System with Low Power Dissipation and High Fidelity*. In Audio Engineering Society Convention 129, Nov 2010. URL http://www.aes.org/e-lib/browse.cfm?elib=15658.

Kazushige Kuroki, Ryota Saito, Naoto Shinkawa, Tomohiro Tsuchiya, and Akira Yasuda. *A Digitally Direct Driven Dynamic-Type Loudspeaker*. In Audio Engineering Society Convention 124, May 2008. URL http://www.aes.org/e-lib/browse.cfm?elib=14474.

Michael Lester and Jon Boley. The effects of latency on live sound monitoring. In *Audio Engineering Society Convention 123*, Oct 2007. URL http://www.aes.org/e-lib/browse.cfm?elib=14256.

Stanley P. Lipshitz and John Vanderkooy. *Digital Dither*. In Audio Engineering Society Convention 81, Nov 1986. URL http://www.aes.org/e-lib/browse.cfm?elib=5018.

Stanley P. Lipshitz and John Vanderkooy. *Why Professional 1-Bit Sigma-Delta Conversion is a Bad Idea*. In Audio Engineering Society Convention 109, Sep 2000. URL http://www.aes.org/e-lib/browse.cfm?elib=9150.

Zhi Liu, Fan Wu, and Qing Yang. *The Training and Analysis on Listening Discrimination of Pure Tone Frequency*. In Audio Engineering Society Convention 122, May 2007. URL http://www.aes.org/e-lib/browse.cfm?elib=14100.

Karl MacMillan, Michael Droettboom, and Ichiro Fujinaga. *Audio Latency Measurements of Desktop Operating Systems*. In Proceedings of the International Computer Music Conference, Sep 2001. URL http://hdl.handle.net/2027/spo.bbp2372.2001.055.

Robert C. Maher. *On the Nature of Granulation Noise in Uniform Quantization Systems*. J. Audio Eng. Soc, 40(1/2):12–20, 1992. URL http://www.aes.org/e-lib/browse.cfm?elib=7062.

Microsoft Corporation. *IAudioClient::GetMixFormat method*. 2012a. URL http://msdn.microsoft.com/en-us/library/windows/desktop/dd370872(v=vs.85).aspx.

Microsoft Corporation. *About the Windows Core Audio APIs*. 2012b. URL http://msdn.microsoft.com/en-us/library/windows/desktop/dd370802(v=vs.85).aspx.

Microsoft Corporation. *IAudioClient::Initialize method*. 2013. URL http://msdn.microsoft.com/en-us/library/windows/desktop/dd370875(v=vs.85).aspx. Last retrieved in the 11$^{\text{th}}$ of April, 2014.

S. K. Mitra. *Digital Signal Processing: A Computer Based Approach*. McGraw-Hill series in electrical and computer engineering. McGraw-Hill Education, 2006. ISBN 007-124467-0.

Christopher Montgomery. *24/192 Music Downloads ...and why they make no sense*. 2013. URL http://people.xiph.org/~xiphmont/demo/neil-young.html. Last retrieved in the 19th of January, 2013.

P. J. A. Naus and E. C. Dijkmans. *Low Signal-Level Distortion in Sigma-Delta Modulators*. In Audio Engineering Society Convention 84, Mar 1988. URL http://www.aes.org/e-lib/browse.cfm?elib=4822.

Katsuya Ogata, Tsuyoshi Soga, Hajime Ueno, and Akira Yasuda. *Digital-Driven Piezoelectric Speaker using Multi-Bit Delta-Sigma Modulation*. In Audio Engineering Society Convention 121, Oct 2006. URL http://www.aes.org/e-lib/browse.cfm?elib=13777.

Roberto Osorio-Goenaga. *Digital Filter Design and Implementation within the Steinberg Virtual Studio Technology (VST) Architecture*. In Audio Engineering Society Convention 119, Oct 2005. URL http://www.aes.org/e-lib/browse.cfm?elib=13313.

Henry W. Ott. *Electromagnetic Compatibility Engineering*, pages 1–43. John Wiley & Sons, Inc., 2009. ISBN 9780470508510. URL http://dx.doi.org/10.1002/9780470508510.ch1. DOI 10.1002/9780470508510.ch1.

PortAudio Community. *PortAudio – Portable Cross-platform Audio I/O*. 2013. URL http://www.portaudio.com/. Last retrieved in the 24th of November, 2013.

PulseAudio Developers. *What Is PulseAudio?* 2013. URL http://www.freedesktop.org/wiki/Software/PulseAudio/. Last retrieved in the 24th of November, 2013.

Bruno Putzeys and Renaud de Saint Moulin. *Effects of Jitter on AD/DA Conversion - Clock and Interface Jitter Specifications*. In Audio Engineering Society Convention 116, May 2004. URL http://www.aes.org/e-lib/browse.cfm?elib=12637.

Realtek Semiconductor Corporation. *ALC885*. 2013. URL http://www.realtek.com/products/productsView.aspx?Langid=1&PNid=24&PFid=28&Level=5&Conn=4&ProdID=138. Last accessed in the 10th of January, 2013.

Derk Reefman and Erwin Janssen. *Enhanced Sigma Delta Structures for Super Audio CD Applications*. In Audio Engineering Society Convention 112, Apr 2002. URL http://www.aes.org/e-lib/browse.cfm?elib=11395.

Alec H. Reeves. *Electric Signaling System*. 1942. Patent. US 2272070. February 3rd, 1942. Filed November 22nd, 1939. Filing number 305665. IPC class G01S 1/00 (20060101), US class 340/870.19.

Joshua D. Reiss. *Understanding Sigma-Delta Modulation: The Solved and Unsolved Issues*. *J. Audio Eng. Soc*, 56(1/2):49–64, 2008. URL http://www.aes.org/e-lib/browse.cfm?elib=14375.

M. P. Robillard. *What Makes APIs Hard to Learn? Answers from Developers*. *Software, IEEE*, 26(6):27–34, 2009. ISSN 0740-7459. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5287006&isnumber=5286993. DOI 10.1109/MS.2009.193.

Thomas D. Rossing, Richard F. Moore, and Paul A. Wheeler. *The Science of Sound*. Addison Wesley, 3rd edition, 2002. ISBN 0-8053-8565-7.

Francis Rumsey. *Blu-ray or Downloads for HD Audio Delivery?* J. Audio Eng. Soc, 59 (3):149–154, 2011. URL http://www.aes.org/e-lib/browse.cfm?elib=15781.

T. Scheller and E. Kuhn. *Influencing Factors on the Usability of API Classes and Methods*. pages 232–241, 2012. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6195191&isnumber=6195161. DOI 10.1109/ECBS.2012.27.

Adel S. Sedra and Kenneth C. Smith. *Microelectronic Circuits*. The Oxford series in electrical and computer engineering. Oxford University Press, Oxford, 5th edition, 2004. ISBN 0-19-514252-7.

Mike Story. *Audio Analog-to-Digital Converters*. J. Audio Eng. Soc, 52(3):145–158, 2004. URL http://www.aes.org/e-lib/browse.cfm?elib=12987.

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley-Longman, 3rd edition, 2005. ISBN 0-20-188954-4.

John Vanderkooy and Stanley P. Lipshitz. *Dither in Digital Audio*. J. Audio Eng. Soc, 35(12):966–975, 1987. URL http://www.aes.org/e-lib/browse.cfm?elib=5173.

Yonghao Wang. *Latency Measurements of Audio Sigma Delta Analog to Digital and Digital to Analog Converts*. In Audio Engineering Society Convention 131, Oct 2011. URL http://www.aes.org/e-lib/browse.cfm?elib=16581.

Yonghao Wang, Ryan Stables, and Joshua Reiss. *Audio Latency Measurement for Desktop Operating Systems with Onboard Soundcards*. In Audio Engineering Society Convention 128, May 2010. URL http://www.aes.org/e-lib/browse.cfm?elib=15378.

Kyosuke Watanabe, Akira Yasuda, Hajime Ohtani, Ryota Suzuki, Naoto Shinkawa, Tomohiro Tsuchiya, and Kenzo Tsuihiji. *A Novel Beam-Forming Loudspeaker System Using Digitally Driven Speaker System*. In Audio Engineering Society Convention 127, Oct 2009. URL http://www.aes.org/e-lib/browse.cfm?elib=15144.