Audio software development –
an audio quality perspective

Jonas Ekeroott

Department of Music and Media
Luleå University of Technology

August 9, 2007
Abstract

Audio Technology as an academic discipline at Luleå University of Technology does not yet have a long research tradition and is therefore in the process of forming and establishing its research areas and methods. In that context, this essay treats audio software development from an audio quality perspective. The aim was to identify relevant questions that must be considered during such software development. Audio aspects of the operating systems Windows, Mac OS X and Linux were included in this essay, and the general development perspective was on audio software written in C++ to be run on general purpose CPUs. A research review, comprising literature from different fields such as Audio Engineering, Computer Science and Software Engineering, was conducted to summarize and integrate an overview of the state of knowledge, and at the same time observe what was still missing in the literature. The result can be viewed as a map of questions that constitute the starting point for future research activities, consisting of further literature studies and experiments with software prototypes. Such software prototypes were not developed during the work with this essay. In summary, questions that must be considered during audio software development from an audio quality perspective deal with the audio software signal path through an audio subsystem, the use of floating point audio sample representation, conversions between floating point and fixed point integer audio sample representation, and if and how to apply dither in the context of floating point audio sample representation.
Keywords

Audio, software, development, API, signal path, C++, floating point, dither, quality
List of Figures

1  The role of a compiler ............................. 10
2  The vague term audio architecture ............... 13
3  General layered audio software structure .......... 15
4  MME/WDM layered audio software structure ........ 20
5  DS/WDM layered audio software structure .......... 21
6  ASIO layered audio software structure ............. 22
7  WASAPI layered audio software structure .......... 23
8  Core Audio and ALSA layered audio software structures ... 23
9  C++ data types – fixed point integer ............... 27
10 C++ data types – floating point .................... 27
11 SNR – fixed point integer versus floating point ... 29
List of Tables

1. Native audio APIs for Windows, Mac OS X and Linux . . . . 19
2. Sizes in bits of $s$, $e$ and $f$ for floating point numbers . . . . 25
3. C++ data types for fixed point integer audio sample values . 26
4. C++ data types for floating point audio sample values . . . . 28
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Audio Engineering Society</td>
</tr>
<tr>
<td>ALSA</td>
<td>Advanced Linux Sound Architecture</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASIO</td>
<td>Audio Stream Input/Output</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DS</td>
<td>DirectSound</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EBU</td>
<td>European Broadcasting Union</td>
</tr>
<tr>
<td>GAE</td>
<td>Global Audio Engine</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>ITU-R</td>
<td>International Telecommunication Union Radiocommunication Assembly</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>LTU</td>
<td>Luleå University of Technology</td>
</tr>
<tr>
<td>MIDI</td>
<td>Musical Instrument Digital Interface</td>
</tr>
<tr>
<td>MME</td>
<td>Multimedia Extensions</td>
</tr>
<tr>
<td>NAMM</td>
<td>National Association of Music Merchants</td>
</tr>
<tr>
<td>PEAQ</td>
<td>Perceptual Evaluation of Audio Quality</td>
</tr>
<tr>
<td>R&amp;D</td>
<td>Research &amp; Development</td>
</tr>
<tr>
<td>SNR</td>
<td>signal-to-noise ratio</td>
</tr>
<tr>
<td>SR</td>
<td>Swedish Radio, Ltd.</td>
</tr>
<tr>
<td>WASAPI</td>
<td>Windows Audio Session API</td>
</tr>
<tr>
<td>WDM</td>
<td>Windows Driver Model</td>
</tr>
<tr>
<td>WMP</td>
<td>Windows Media Player</td>
</tr>
</tbody>
</table>
1 Introduction

Three quotes contributed to the ideas that led further to the work performed by the author, and which is presented in this essay. Michael Kelly, cochair of the AES Technical Committee on Audio for Games, audio programmer at Sony Computer Entertainment Europe, and PhD in spatial sound and audio coding at the University of York in 2002, wrote in the article *Using game-audio tools to build audio research applications* [1] in the November 2006 issue of the Journal of the Audio Engineering Society:

“The process of getting audio in and out of the computer is not always clear to the uninitiated, even to an otherwise experienced programmer.”

He also noted:

“Audio researchers often require custom software tools in order to perform their day-to-day tasks. While the number of off-the-shelf tools continue to grow, there are many applications with specific requirements that cannot be satisfied by widely available tools.”

Jan Berg, Senior Lecturer in Audio Technology at the Department of Music and Media, Luleå University of Technology, wrote the following introduction on audio quality in audio work in his PhD thesis [2]:

“Evaluation of the audio quality a listener perceives is an important issue for all those involved in audio recording and reproduction. The variety of tools in audio work facilitates advanced processing of the audio signal. The aims of these processes differ, but they all in some way, sooner or later give rise to the question of in which way the audio quality is influenced by them.”

These quotes coupled with the author’s own experience from working with audio quality and audio software issues for many years as a Research Engineer at Swedish Radio (SR), Research & Development (R&D), led to a deeper interest into how the choice of different ways to implement audio software on different operating system platforms can have an influence on the audio quality of such audio software.

1.1 Essay aim and limitations

The aim of this essay was to identify questions that have to be asked and carefully considered in audio software development from a high audio quality perspective. An example of such a scenario is the development of listening test software in audio research, on a small non-commercial scale. In such a context, an optimal technical audio quality is strived for in the sense
that no unknown or uncontrolled processing or degradation of the audio signal has occurred, and consequently that no audio quality degradation can be perceived or that it is minimized. A rather different meaning of audio quality can be given in a home user environment where ease of use, e.g. when listening to one’s favourite music as MP3 files, generally is more important than having to worry about sampling rates and format conversions. The listening in such a situation is for pleasure, and unknown processing or degradation can probably be accepted in most cases. In this essay however, audio quality should be understood in the former professional sense above.

Literature concerning audio software development from an audio quality perspective is practically non-existent. Therefore, the author saw a need to contribute in this area and to write an initial text on the subject. The result is this essay. It might presumably be the case that such kind of knowledge already exists within commercial software development companies as private and protected knowledge. However, it is the author’s experience from international R&D work for SR that these kinds of issues that this essay puts in writing, do not have a widespread clear understanding within the audio technology community, including commercial companies. Thus, even if the knowledge, or parts of it, already exists inaccessible inside software companies, there is a need for a free, open and accessible text that treats such issues.

The target platforms considered in this essay were limited to Windows, Mac OS X and Linux, as a choice of one of them is common for running both general audio production software as well as specialized audio research software. The perspective of development in the essay was on software written in high-level C++, using an audio API (Application Programming Interface) provided by the operating system platform, to be run on a general purpose CPU (Central Processing Unit), i.e. not software written in low-level assembler to be run on a specialized DSP chip. Taking C++ as a basic condition for this study does not mean that lots of C++ code were developed and are included in the essay, but rather that the overall context of the discussion is assumed to be C++, since this is the most common audio programming language.

No experimental evaluations of audio quality in audio software were performed in this essay. Details about the exact audio software signal path through a computer audio system still remain difficult to shed light on and unambiguously describe. Without such detailed knowledge, experiments, e.g. listening tests, risk to contain too many uncontrolled technical factors to provide reliable results. The work in this essay was therefore instead theoretically focused on trying to clear up the field as a basis for future deepened and experimental studies with software prototypes.

Knowledge of Audio Technology is assumed, and some familiarity with basic concepts of general computer terminology and software development is probably helpful when reading this essay.
2 Background

2.1 Software development

Making a piece of software involves writing the source code in for example C++ in one or more files, and then compiling the source code into an executable program for a specific CPU, as shown in figure 1. In an audio context, the executable program would typically be called an audio application (see terminology in section 2.2 below).

Figure 1: The role of a compiler.

The compiler itself is also an executable program.

The classic waterfall model for software development was given by Royce [3]. The model comprises a number of phases through which the development process sequentially flows. In reality though, it is often an iterative process through the phases, as Royce also pointed out. Wiktorin [4] described a slightly condensed and simplified version of the model, with the following four phases:

- requirements specification
- analysis
- construction
- testing

A good requirements specification is of crucial importance for a successful software development project. Finding the requirements and formulating them in an unambiguous and understandable way is not a trivial task. This
is the case for software development in general, and particularly for audio software development (compare the discussion in section 6.1).

In the analysis phase the requirements specification, which is often expressed in rather sweeping words, is transferred into precise descriptions of data and its associated processing in the software system under development.

During the construction phase, which is alternatively called implementation or coding, the results from the analysis phase are transferred into source code written in a programming language, like e.g. C++.

Finally, the testing phase uses carefully selected test cases to evaluate the performance of the developed software, in order to validate that it meets the given requirements. A distinction is made between black box testing in which no information about the source code or internal structure of the software is known, and white box testing in which such information is available and can be used in the definition of test cases.

For the development of audio software with the highest audio quality requirements, all these four phases should be carried out with a considerable element of professional audio production knowledge involved. This is further discussed in chapter 6 of this essay. In section 6.5 on future work is also mentioned the need for establishing a set of audio software quality metrics as an aid in the evaluation of audio quality in audio software development.

### 2.2 Terminology for audio software development

The terminology within the field of audio software development is inconsistent in the sense that various meanings can be given to a specific word. Examples of confusing use of terminology are given after the definitions below. The meaning can also differ due to differences from the perspective of multi-platform development, e.g. between the target platforms Windows, Mac OS X and Linux. Yet, a uniform multi-platform terminology, free from uncritical acceptance and usage of vague terms, would be very valuable as a basis for unambiguous communication within the field. This is the author’s own experience from years of work at Swedish Radio R&D, including international contacts within the audio technology community.

As a remedy to this situation, in this essay the following definitions apply and are proposed to be used as a more uniform terminology. Figures 2 and 3 give an illustration of the relationships of some of the terms. The terms are given in a conceptual, i.e. not alphabetical, order.

**computer audio system** A computer with some form of installed audio hardware for audio input/output (I/O), e.g. a sound card or an external audio interface, and audio software for recording, mixing, editing, etc. Normally no ambiguity in the term, but included for completeness.
**user mode** Machine instructions pertaining to audio applications (see below) run on the CPU in this mode [5], where simultaneously running applications are protected from interfering with each other’s resources, e.g. memory areas, to avoid application and operating system crashes. Normally no ambiguity in the term from an audio perspective, but needed for following definitions. See figure 2.

**kernel mode** Machine instructions pertaining to the operating system run on the CPU in this mode [5]. Such code has more unrestricted access to resources in the computer audio system. Typically better performance than for user mode software. Normally no ambiguity in the term from an audio perspective, but needed for following definitions. See figure 2.

**audio application** User mode software, normally with a Graphical User Interface (GUI) for human user interaction like recording, mixing, editing, etc. See figures 2 and 3.

**native audio API** An audio Application Programming Interface (API) that defines and provides the high-level, e.g. C++, functions that an audio application can call to get audio functionality, like communicating with installed audio hardware. Implemented in user mode. Normally proprietary. A “native” audio API is one that is provided by the operating system on the respective platform. One or several native audio APIs can exist for the same operating system. See figure 3 and section 4.1.2.

**multi-platform audio API** An audio API that is layered on top of a number of native audio APIs on multiple platforms. Implemented in user mode. Normally open source. Is dependent on and calls audio functions provided by the underlying native audio API. See figure 3. Examples include PortAudio\(^1\), OpenAL\(^2\) and JACK\(^3\), but they are not discussed further in this essay, which is restricted to audio software development from an audio quality perspective using native audio APIs.

**driver** Kernel mode software that communicates with the underlying audio hardware on a low machine level. Provides a level of abstraction that conceals hardware level details for a specific user mode native audio API that is layered on top of the driver. See figure 3.

**audio architecture** A vague term that should be used with caution and, if used, be supplemented with a clarification of what is exactly meant.

---

\(^1\)http://www.portaudio.com/
\(^2\)http://www.openal.org/
\(^3\)http://jackaudio.org/
Refers to user and kernel mode software that, in addition to an audio application, native audio API and driver, are part of the audio handling in an operating system. See figure 2 and section 4.1.3.

**audio subsystem** A broad term for the parts of a computer audio system that deals with audio functionality. Proposed to replace the term audio architecture. The word *architecture* easily leads the thought to hardware related issues (e.g. CPU instruction set, memory organization, input/output devices and bus structure [5]). The word *subsystem* hopefully could aid in the understanding that, apart from the audio hardware, user and kernel mode *software* are the issue at hand.

**audio engine** A very vague term that should be avoided altogether. Different known uses include software in various layers in figure 2 and 3.

**audio software signal path** The *total* audio signal path through *all* the user and kernel mode *software* in an audio subsystem in a computer audio system, including operating system software.

This list of terminology should not be viewed as being absolutely complete and exhaustive. It is a first attempt to present such terminology in print in one place. Future work will possibly make changes, additions and refinements to the list.

![Figure 2: The vague term audio architecture illustrated as a cloud somewhere in and/or between an audio application and an operating system. After Ekeroot [6].](image)

Examples of confusing use of terminology are not easy to give in a condensed manner, without having to introduce an additional and quite exten-
sive operating system specific technical terminology. An attempt to illus-
trate the situation though, which led to the formulations in the terminology
given above, is the following paragraphs.

In many contexts the notion audio API is used without a clear distinction
between native and multi-platform. This fails to explicitly point out their
hierarchical relation and that in the case of a multi-platform audio API, it
is layered on top of a native audio API. Thus, they both need to be taken
into account when trying to build a clear understanding of the software
components involved in the audio software signal path. Just considering
a multi-platform audio API alone disregards the details of the underlying
native audio API layer. An example is in the article on building audio
research applications [1], mentioned in the introduction of this essay, in the
treatment of the multi-platform audio API OpenAL (see the terminology
above).

The software company Cakewalk invited representatives from Microsoft
and more than 30 hardware and software companies to a Windows Profes-
sional Audio Roundtable at the February 2000 NAMM (National Association
of Music Merchants) trade show. The purpose was to collaborate to make
the Windows platform an ideal choice for professional audio. In a resulting
white paper [7], under Observation 3: The term “driver” is misunderstood,
they noted a confusion between drivers and native audio APIs:

“A true ‘driver’ runs in the kernel … Technologies like MME,
ASIO … are merely user-mode APIs, not drivers.”

User mode native audio APIs like MME and ASIO are further treated in
section 4.1.2.

In section 4.1.3 on audio software signal paths associated with the use
of a specific native audio API, figures 4 and 5 both illustrate what is com-
monly referred to as the WDM Audio Architecture, where WDM stands for
Windows Driver Model. The notion architecture in this case takes into ac-
count the total audio software signal path. In contrast, another Microsoft
notion, the Universal Audio Architecture (UAA)\(^4\) associated with the Win-
dows Vista operating system and not further discussed in this essay, is de-
scribed as a class driver architecture that provides basic audio functionality
for compliant audio hardware. It thus essentially refers to a generic audio
driver, i.e. not a total audio software signal path but only the driver layer
of it.

The audio software signal path shown in section 4.1.3 in figure 7 contains
a user mode software component called Global Audio Engine (GAE). It is
provided by the operating system and is a central part of the audio software
signal path, common to many different audio applications. An example of
a different use of the notion audio engine is the file AudioEngine.dll in

\(^4\)http://www.microsoft.com/whdc/device/audio/uaa.mspx
the plug-ins folder of the WaveLab\textsuperscript{5} 4.0 installation on the author’s laptop Dell Latitude D600. This is a user mode software component related to a specific audio application, in this case WaveLab, and not a generic part of the audio subsystem of an operating system platform.

2.3 General layered audio software structure

As a condensed summary of the previous section, a formal view of the general layered audio software structure in a computer audio system is shown in figure 3:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3.png}
\caption{A formal view of the general layered audio software structure in a computer audio system, including the possible use of a multi-platform audio API.}
\end{figure}

The inclusion of a multi-platform audio API, which is optional, adds to the complexity of the audio software signal path since a multi-platform audio API is dependent on audio functions provided by the underlying native audio API. Though, for multi-platform audio applications, using a multi-platform audio API can greatly facilitate and speed up development, since only one API has to be studied and learned instead of a number of platform specific native audio APIs.

\textsuperscript{5}http://www.steinberg.net/128_1.html
3 Methodological considerations

Audio Technology as an academic discipline at Luleå University of Technology (LTU) does not yet have a long research tradition and is therefore in the process of forming and establishing its research areas and methods. At LTU Audio Technology is defined as the study of processes and methods for

- recording
- processing
- reproduction

of sound from both a technical and artistic point of view. Methods for the evaluation of audio quality, e.g. by means of listening tests with human listeners, has so far been the main area of research. This essay has a different focus and is technical in its approach, with a specific orientation towards issues in audio software development that can have an influence on audio quality. The author thus hopes to have been contributing to the advancement and broadening of the emerging research field of Audio Technology at LTU.

3.1 Research review

The model for the research process for this essay was what Backman [8] called research synthesis (in Swedish “forskningsöversikt”), i.e. a summarizing and integrating overview of the state of knowledge, in this case within the field of audio software development from an audio quality perspective, through a literature review. An alternative English term would be research review. The motivation for this approach – to make a research review that results in questions for future studies – was that the topic of the essay constitutes a new subfield within the subject of Audio Technology at LTU, with little written about it.

3.2 Finding relevant literature

As noted in the introduction of this essay, literature that treats audio software development from an audio quality perspective is practically nonexistent. For example, the Audio Engineering literature traditionally does not treat audio software development. Fragments of useful information therefore had to be looked for in literature from other disciplines such as Computer Science and Software Engineering. Electronic databases searched for information during the work with this essay include the Audio Engineering Society, Compendex and Inspec.
Also on-line sources were searched and studied. See section 4.1.3 and associated footnotes.

Audio related open source code projects can also be considered as literature within the field of audio software development. Major projects include CLAM\textsuperscript{6}, SndObj\textsuperscript{7}, Audacity\textsuperscript{8} and Ardour\textsuperscript{9}. In contrast to commercial software, such projects provide access to the source code, often in C++, which can be studied in detail to find out the used coding practices, and then modified for further experimentation and prototype development. The ease with which such open source code can be studied and understood depends highly on how well structured and commented the source code is, and in any case the effort required to gain both an overall and a detailed understanding of such projects is normally quite time consuming. Therefore, the detailed study of open source code projects had to be left outside the scope of this essay, and remain for future research activities.

3.3 A professional audio perspective

The search for relevant literature and the following evaluation and analysis of it was guided by the author’s several years of experience as a Research Engineer at Swedish Radio, including international work within the European Broadcasting Union (EBU), with issues concerning audio quality and audio software. From this professional audio perspective the literature was critically surveyed to summarize and integrate an overview of the state of knowledge, and at the same time to observe what was still missing within the field of audio software development from an audio quality perspective.

3.4 Strengths and weaknesses

A consequence of the choice of basing this study on a literature review even though there was very little literature within the field, was that the input of ideas and results from other researchers was rather limited, which could be claimed as a weakness. The author’s own experience thus got a prominent position in this study with limited possibilities of comparisons to other findings. On the other hand, a strength in this choice was that the focus was really put on the literature review and the author’s attention was not distracted by e.g. software prototype development in parallel. Since this was an early study within its field, a research review that collected and reported initial literature findings was a paramount consideration.

\begin{itemize}
\item \textsuperscript{6}http://clam.iua.upf.edu/
\item \textsuperscript{7}http://sndobj.sourceforge.net/
\item \textsuperscript{8}http://audacity.sourceforge.net/
\item \textsuperscript{9}http://ardour.org/
\end{itemize}
4 Audio software development

This chapter treats distinguishing characteristics for audio software development compared to non-audio software development.

4.1 Audio subsystem

An evident major distinguishing characteristic for an audio application is the input/output of audio samples by means of communication with some audio hardware, e.g. a sound card or an external audio interface. The following sections treat software aspects of the audio subsystem involved in this communication.

4.1.1 Audio input/output

The classic organization of any computer into five fundamental components was described by Patterson and Hennessy [9]:

- input
- output
- memory
- datapath
- control

The last two, datapath and control, are usually collectively called the Central Processing Unit (CPU). Audio issues regarding the CPU are discussed in section 4.2 below in the treatment on floating point. Memory is needed by all software, audio as well as non-audio, and was excluded from further treatment in this essay.

Input and output (I/O) in the context of a typical audio application were discussed by Amatriain [10], and are summarized in the following list:

- sound card input
- sound card output
- sound file I/O
- compressed sound file I/O
- MIDI I/O
- network I/O
A brief remark on audio issues regarding sound file I/O is in section 4.2.3 below in the treatment on floating point. Compressed sound file I/O (e.g. MP3), MIDI I/O and network I/O were out of the scope for this essay, which consequently was restricted to the first two items – alternatively referred to as audio I/O.

The actual audio hardware that takes care of the audio I/O includes circuits for analogue/digital (A/D) and digital/analogue (D/A) conversion. Their associated electrical characteristics and objective quality measurements, which were discussed in detail by Jones et al [11] and in the AES Information document for digital audio – Personal computer audio quality measurements AES-6id-2006 [12], were also left outside the software focused treatment in this essay.

As already shown in figure 3 on page 15, the audio hardware used for audio I/O is accessed by an audio application by means of the functionality provided by a native audio API.

### 4.1.2 Native audio APIs

Ekeroot [6] gave an overview of current native audio APIs for Windows, Mac OS X and Linux, which is summarized in the following table:

<table>
<thead>
<tr>
<th>platform</th>
<th>native audio API</th>
<th>company</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>MME (Multimedia Extensions)</td>
<td>Microsoft</td>
</tr>
<tr>
<td></td>
<td>DS (DirectSound)</td>
<td>Microsoft</td>
</tr>
<tr>
<td></td>
<td>WASAPI (Windows Audio Session API)</td>
<td>Microsoft</td>
</tr>
<tr>
<td></td>
<td>ASIO (Audio Stream Input/Output)</td>
<td>Steinberg</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>Core Audio</td>
<td>Apple</td>
</tr>
<tr>
<td>Linux</td>
<td>ALSA (Advanced Linux Sound Architecture)</td>
<td>open source</td>
</tr>
</tbody>
</table>

**Table 1:** Current native audio APIs for Windows, Mac OS X and Linux.

All of the native audio APIs in the table are proprietary, as indicated by the third column, except ALSA which is open source\(^\text{10}\). WASAPI is the

\(^{10}\) http://www.alsa-project.org/
latest native audio API by Microsoft and thus only available in Windows Vista.

Associated with the use of a specific native audio API from table 1, is a set of user and kernel mode software components that in a hierarchical fashion constitutes the audio software signal path. Illustrations of such audio software signal paths are shown in the next section.

### 4.1.3 Audio software signal path

Figures 4 to 8, structured in the same way as figures 2 and 3 on page 13 and 15, were presented by Ekeroot [6]. They are an initial attempt to illustrate the audio software signal path in a summarized way for the native audio APIs that are listed in table 1. The figures were theoretically derived by extracting and compiling information from Microsoft Developer Network\(^\text{11}\), Steinberg 3rd Party Developers\(^\text{12}\), Apple Developer Connection\(^\text{13}\) and Advanced Linux Sound Architecture – ALSA\(^\text{14}\). Hitherto the figures are not verified or analyzed from the perspective of audio quality by software experiments, but this is left for future studies.

Figure 4 shows the many software layers involved for an MME audio application under the current Windows Driver Model (WDM).

![Diagram of MME/WDM layered audio software structure](image)

**Figure 4:** MME/WDM layered audio software structure, e.g. on Windows XP. After Ekeroot [6].

---

\(^\text{11}\)http://msdn.microsoft.com/
\(^\text{12}\)http://www.steinberg.net/324_1.html
\(^\text{13}\)http://developer.apple.com/
\(^\text{14}\)http://www.alsa-project.org/
Green colour indicates software by an audio application company, yellow colour indicates software by Microsoft and pink colour indicates software by an audio hardware company. Audio propagates through the stack by a software component in one layer calling functions in a software component in another layer below or above. The textual descriptions to the right of the rectangles, extracted from the actual files, are included as an illustration of the problem that such names normally do not give any help in understanding the hierarchical position or purpose of a given software component. The stac97.sys file is an example from the author’s laptop Dell Latitude D600. In the middle of the stack is the kernel mode audio mixer kmixer.sys which can perform audio sample format and sample rate conversion. The only control of this mixer is via a control panel, mmsys.cpl, with a GUI slider to adjust the quality of the sample rate conversion in three positions – Good, Better, Best – without further indication of conversion details.

The layered structure for a DirectSound (DS) audio application, e.g. Windows Media Player (WMP) under WDM is shown in figure 5. It contains somewhat fewer layers than the MME/WDM case in figure 4. Depending on specifications of the audio hardware, the kernel mode audio mixer can either be in or out of the audio software signal path, but this is out of control from the perspective of the source code in the uppermost audio application layer.

![Layered Audio Software Structure](image)

**Figure 5:** DS/WDM layered audio software structure, e.g. on Windows XP. After Ekeroot [6].

Figure 6 illustrates how an ASIO audio application bypasses all the Microsoft software layers in the MME/WDM and DS/WDM cases in figures 4 and 5, including the kernel mode audio mixer. The ASIO native audio API is provided by Steinberg, indicated by blue colour in the figure.
The latest version of the Microsoft operating system, Windows Vista, introduced WASAPI – the Windows Audio Session API. As seen to the left and in the middle of figure 7, the layered audio software structure can still include MME and DS but layered on top of WASAPI. Most of the kernel mode software components in figures 4 and 5, except for the WDM part at the bottom of the stack, have been removed and replaced with a new so called Global Audio Engine (GAE) in user mode, with mixing functionality reminiscent of the kernel mode audio mixer in the MME/WDM and DS/WDM cases. The internal audio sample representation format is changed from 16-bit fixed point integer to 32-bit floating point (see section 4.2 below). There is a possibility to bypass the Global Audio Engine altogether by using the WASAPI native audio API directly, as shown to the right in figure 7.

Finally, as an initial comparison to the above Windows cases which for time priority reasons were given the majority of the author’s attention, figure 8 outlines the Core Audio and ALSA cases for Mac OS X and Linux respectively. Details need to be filled in by future studies. Blue colour indicates software by Apple. The uncoloured boxes in the ALSA case indicate that software internals can be analyzed in detail through the access to open source code.

In addition to the figures presented here, other attempts to illustrate audio software signal paths in a computer audio system were made in the AES Information document for digital audio – Personal computer audio quality
Figure 7: WASAPI layered audio software structure on Windows Vista. After Ekeroot [6].

Figure 8: Core Audio and ALSA layered audio software structures on Mac OS X and Linux respectively. After Ekeroot [6].
measurements AES-6id-2006 [12] and by Jones et al [11]. These documents contain simplified signal path diagrams solely for a general Windows case, without any specific indication of the intended native audio API. Nor is any information about the actual user and kernel mode software components given, as on the contrary is done in the figures presented above in this essay.

4.1.4 Level diagram

An interesting, and from an audio quality perspective much needed, complement to a figure of a particular audio software signal path, as shown in figures 4 to 8 above, would be a level diagram indicating nominal level, maximum allowed level before clipping and noise floor. Jones et al noted [11]:

“This is an area of difficulty in PC audio devices; there is seldom an adequate indication of signal level at any part of the signal path.”

Such level diagrams are common for traditional analogue and digital audio equipment, but so far lacking in the context of computer audio subsystems.

4.2 Floating point

Goldberg stated the following in the paper What every computer scientist should know about floating-point arithmetic [13]:

“Floating-point arithmetic is considered an esoteric subject by many people. This is rather surprising, because floating-point is ubiquitous in computer systems.”

Further, Knuth [14] noted:

“...every well-rounded programmer ought to have a knowledge of what goes on during the elementary steps of floating point arithmetic. This subject is not at all as trivial as most people think; it involves a surprising amount of interesting information.”

Floating point audio sample representation is very common in current audio applications since it provides large headroom and large dynamic range in intermediate signal processing (see figures 10 and 11 below).

4.2.1 Formats

Computer audio systems running either Windows, Mac OS X or Linux and using general purpose CPUs from e.g. Intel, handle non-integer numbers according to the IEEE Standard for binary floating-point arithmetic ANSI/IEEE Std 754-1985 [15]. The interesting floating point formats from a C++ perspective are (C++ data types given in parenthesis):
- 32-bit single format (float)
- 64-bit double format (double)
- 80-bit double extended format (long double)

A floating point number is divided into three components:

- sign (s)
- exponent (e)
- fraction (f)

The sign bit $s$ is 0 for positive numbers and 1 for negative numbers, the exponent $e$ is an unsigned fixed point integer number, and the bits in the fraction $f$ represent $2^{-1}$, $2^{-2}$, $2^{-3}$, etc, counting from the leftmost most significant bit. The following table gives the sizes in bits of each component for the three formats listed above:

<table>
<thead>
<tr>
<th>format</th>
<th>sign</th>
<th>exponent</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>1</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>double</td>
<td>1</td>
<td>11</td>
<td>52</td>
</tr>
<tr>
<td>double extended</td>
<td>1</td>
<td>15</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 2: Sizes in bits of $s$, $e$ and $f$ for floating point numbers.

Only the 32-bit single format is further treated in the following text. With a few exceptions, e.g. to represent $\pm\infty$, the value $v$ of a 32-bit single format number is given by the formula

$$v = (-1)^s 2^{e-127}(1.f)$$

and such floating point numbers are called normalized. The formula has an implicit one, i.e. $2^0$, preceding the implied binary point to the left of the fraction bits. Denormalized numbers are very small numbers close to zero. Their value is given by the formula

$$v = (-1)^s 2^{-126}(0.f)$$

and they are identified by having $e = 0$. They do not have an implicit one preceding the implied binary point to the left of the fraction bits in the formula.
4.2.2 Denormalized numbers

Schwarz et al [16] made the following remark on denormalized floating point numbers:

“Denormalized numbers are the most difficult type of numbers to implement in floating-point units. They are so complex that some designs have elected to handle them in software rather than hardware. This has resulted in execution times in the tens of thousands of cycles, which has made denormalized numbers useless to programmers.”

De Soras [17] discussed the “denormal bug” and its associated annoying calculation slowness. He reported:

“. . . a multiplication with a denormal operand takes about 170 cycles, which is more than 30 times slower than with normal operands only! When almost all the processed numbers are denormal, CPU load increases a lot and can affect the stability of real-time applications, even on the fastest systems.”

As a solution he proposed eliminating denormalized numbers by setting their values to zero. From a theoretical audio quality perspective this should be plausible since the denormalized numbers start several hundred dB below the level normally taken to represent 0 dB full scale in floating point audio sample representation (compare section 4.2.4 and figure 11 below).

4.2.3 C++ data types

The data types available for audio sample representation to a developer of audio applications in C++, and the associated maximum and minimum audio sample values, are summarized in figures 9 and 10, which were presented by Ekeroot [6].

For fixed point integer values, the available C++ data types are short and int, with the following characteristics including typical maximum and minimum (i.e. 0 dB full scale) audio sample values:

<table>
<thead>
<tr>
<th></th>
<th>maximum</th>
<th>minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>+ 32 767</td>
<td>- 32 768</td>
</tr>
<tr>
<td>int</td>
<td>+ 2 147 483 647</td>
<td>- 2 147 483 648</td>
</tr>
</tbody>
</table>

Table 3: C++ data types for fixed point integer audio sample values.

C++ does not have a separate 24-bit fixed point integer data type, so 24-bit fixed point integer audio sample values must be handled using the 32-bit
Figure 9: C++ data types – fixed point integer. After Ekeroot [6].

Figure 10: C++ data types – floating point. After Ekeroot [6].
The values are asymmetrically distributed around zero, i.e. there is one more value on the negative side, due to the use of two’s complement representation for negative numbers. The values also span the whole available number range meaning that there is an absolute clip limit at the extreme values, indicated by the two long dark areas on the vertical axis in figure 9.

For floating point values, the available C++ data types are `float`, `double` and `long double`, with the following characteristics including typical maximum and minimum (i.e. 0 dB full scale) audio sample values:

<table>
<thead>
<tr>
<th>number of bits</th>
<th>maximum</th>
<th>minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32</td>
<td>+ 1.0</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>+ 1.0</td>
</tr>
<tr>
<td>long double</td>
<td>80</td>
<td>+ 1.0</td>
</tr>
</tbody>
</table>

Table 4: C++ data types for floating point audio sample values.

The values are in this case symmetrically distributed around zero, and on the bit pattern level there are actually two different representations for zero, +0.0 and -0.0, but they are normally both treated as zero. The audio sample values only span the number range [-1.0,+1.0] meaning that there is plenty of headroom for stronger signal levels in intermediate calculations, indicated by the two much shorter dark areas on the vertical axis in figure 10 compared to figure 9. In the end though, the signal level has to be brought back into the range [-1.0,+1.0], e.g. for file storage or output.

The conversion back and forth between 16-bit `short`s, e.g. audio sample values from a WAV file, and 32-bit `float`s, e.g. internally in a typical audio application, was discussed by Erik de Castro Lopo\(^\text{15}\), author of the C library `libsndfile` for reading and writing audio files. Conversion from `short` audio sample values in the range [-32768,+32767] to `float` by dividing by 32768 yields floating point values in the range [-1.0,+0.999969482421875]. Conversion from `float` audio sample values in the full valid range [-1.0,+1.0] back to `short` by multiplying by 32767, yields fixed point integer values in the range [-32767,+32767]. Dividing by one value, 32768, in one direction, and then multiplying by another value, 32767, in the other direction means that this conversion process will not be bit transparent and hence change some audio sample values. De Castro Lopo argued for minimizing the number of conversions to maintain audio quality, and use `short`s only as a final destination storage format and use `float`s for all audio sample values during processing and also intermediate audio file storage.

\(^{15}\text{http://www.mega-nerd.com/libsndfile/FAQ.html}\)
4.2.4 Signal-to-noise ratio

The nature of floating point numbers with the exponent scaling the actual value of the fraction, means that as the exponent takes on smaller values and the resulting floating point numbers approach zero, the gap between consecutive numbers gets smaller, i.e. the precision increases. In contrast, the gap size for fixed point integer numbers is always the same size, i.e. 1. Mathematical derivations of the resulting signal-to-noise ratio (SNR) for both cases were given by Zölzer [18] and also treated by Muheim [19]. Ekeroot [6] presented a summarizing illustration of this, which compares the SNR for 16-bit fixed point integer short:s and 32-bit floating point float:s. This is shown in figure 11.

![Graph showing SNR comparison between 16-bit fixed point integer and 32-bit floating point numbers.](image)

**Figure 11:** SNR – fixed point integer versus floating point. After Ekeroot [6].

The SNR in the fixed point case gradually decreases with signal level. In the floating point case, as the signal level decreases, so does the noise floor for every decreasing step of the exponent value. The resulting SNR graph therefore takes on a sawtooth shape that fluctuates between 144 dB and 138 dB. When the smallest exponent value has been reached and the floating point numbers are denormalized, the SNR behaves as in the fixed point case, but this is several hundred dB down from the “0 dB full scale” level. The lower part of figure 11 also illustrates the large dynamic range in the floating point case, as well as the headroom available since typical audio sample values in the range [-1.0,+1.0] only occupy a part of the total available floating point range (compare figure 10 above).
4.2.5 Dither

Dither in a floating point context is scarcely treated in the general dither literature, where digital dither for requantization is explicitly or implicitly assumed to be fixed point. A thorough theoretical treatment of the mathematical theory of dither as a means for eliminating signal dependent quantization errors was pursued by Wannamaker in his PhD thesis *The theory of dithered quantization* [20], but it did not specifically treat floating point though.

Dunay *et al* [21] noted:

“With the spreading of floating-point DSP’s and IEEE compatible computers, floating-point number representation is more and more widely used. Its quantization error is usually very small, but sometimes it is not negligible. Then, it is justified to add some dither.”

They further proposed:

“The dither for floating-point numbers is preferably a uniform or a triangular-shaped one FOR THE MANTISSA, with the same exponent as of the numbers.”

Here mantissa is the same as fraction in the treatment in section 4.2.1 above.

A consequence of the last quote above is that the dither becomes correlated to the audio signal, i.e. the amplitude of the dither is directly related to the amplitude of the audio signal.

Aldrich [22] argued against such dither:

“Because the dither signal is correlated to the signal itself it is not actually noise and is actually distortion.”

He further noted that adding an uncorrelated constant level dither does not adequately dither the signal. Either the dither level is too high so that the fine low level audio signal details, that are possible due to the large dynamic range of floating point numbers, are eliminated, or the dither level is too low so that most audio sample values will be truncated just as if dither was not added at all. Aldrich concluded:

“...it is simply impossible to adequately dither a floating-point system such that quantization error becomes exclusively quantization noise, such as can be accomplished in a fixed-point system.”

and

“...it is indeed possible that the available dither options will yield less desirable results than removing dither altogether. This
is especially true as programmers try to independently find the
most effective solution, or as they errantly apply the principles
of dither in fixed-point systems to the floating-point domain."

He also pointed out that a distinction has to be made between dithering
and truncating e.g. a 64-bit floating point number into a 32-bit floating point
number, versus dithering and truncating e.g. a 32-bit floating point number
into a 16-bit fixed point integer number. The latter process typically also
involves a scaling (compare section 4.2.3 above).

Further studies in this area need to incorporate evaluations of the audio
quality of using or not using some form of floating point dither, meticulously
described, since a purely theoretically based solution is not possible.

4.3 Audio quality

In audio software development it is important to be able to somehow evaluate
the audio quality of audio applications and also complete audio subsystems
to test and verify that no unknown or uncontrolled processing or degrada-
tion of the audio signal occurs. Based on the treatment in previous sections
in this chapter, such degradation could be the result of audio sample losses
caused by high CPU load due to denormalized floating point numbers, conver-
sions between fixed point integer and floating point numbers, the use of
dither or not in the context of floating point numbers, and possible sample
rate conversion in a software component provided by the operating system
like the kernel mode audio mixer in section 4.1.3. Evaluation of this type of
audio quality can be done from some different viewpoints.

4.3.1 Bit transparency

To verify that no hidden processing or conversion occurs in any part of an
audio subsystem, a test for bit transparency could be used. This would
involve somehow and somewhere in an audio software signal path, as exem-
plified in section 4.1.3 above, inserting a known bit stream and then in a
definite point in the signal path extracting the bit stream and making a bit
wise comparison. If such a test would not turn out to be bit identical, further
objective measurements and perceptual evaluations would be needed.

No such tests were found in the literature.

4.3.2 Objective measurements

As already mentioned in section 4.1.3 above, the Audio Engineering Society
published a document with the title *AES Information document for digital
audio – Personal computer audio quality measurements* AES-6id-2006 [12]
to address issues concerning objective measurements like frequency response
and total harmonic distortion in a computer audio system context. One
restriction with this document is that it only refers to the Windows platform, and a useful extension would be to include also Mac OS X and Linux. Admittedly, this is not an easy task to achieve as it requires extensive multi-platform knowledge and experiences. Another restriction is that the document does not take into consideration the various audio software signal paths that result from using different native audio APIs.

### 4.3.3 Perceptual evaluation

Carefully controlled objective measurements, based on a detailed description of the total audio software signal path, can provide a technical audio quality indication of a computer audio system. Another perspective though is how this quality will be perceived by a human listener. An objectively measured audio quality degradation might or might not be audible, depending on e.g. the order of magnitude of the degradation. The article *Can you really hear it? Psychoacoustics in action* [23] in the January/February 2007 issue of the Journal of the Audio Engineering Society discussed such psychoacoustic aspects, and remarked:

“So it is useful to know either way, whether a supposed improvement in quality is audible or whether a supposed degradation in quality is audible.”

Muheim [19] used the ITU-R PEAQ (Perceptual Evaluation of Audio Quality) method [24] to objectively measure the perceptual effect of a proposed sample loss concealment algorithm in the context of a research project developing a computer audio system. By this he quantified the merits of his proposed algorithm. The PEAQ method gives quality measures that are comparable to the quality measures from the subjective listening test method ITU-R BS.1116 [25] in which a group of listeners assess the perceived audio quality and the mean value of their given grades is calculated as a resulting quality measure. A comparative test using the BS.1116 method would have been interesting to verify the PEAQ results, but was not carried out in his work.

Apart from that study, no other reports of perceptual evaluations of computer audio systems were found in the literature.
5 Results

The research review on audio software development from an audio quality perspective conducted in this essay, was done in order to identify questions that need careful attention and that are still unanswered in the literature. Based on the treatment in previous chapters, these resulting questions are presented below.

5.1 Questions for audio software development

What is the exact audio software signal path, including all user and kernel mode software components, through an audio subsystem using a specific native audio API?

Initial and in some cases simplified overviews are presented in the literature, but details remain to make exhaustive descriptions.

How can such an audio software signal path be unambiguously elucidated and visualized?

An efficient way to achieve this is of utmost importance as a basis for further studies and experimentation.

What is the internal audio sample format of each user and kernel mode software component in such an audio software signal path?

This is important information to know in order to build an understanding of the possible format conversions involved in the audio software signal path.

How can a level diagram for an audio subsystem, in particular the audio software signal path, be established?

The literature mentions the difficulty of this area and the lack of indication of signal level in the audio software signal path, but no solution is proposed.

How should the conversion of audio sample values, in various combinations of bit sizes, from fixed point integer numbers to floating point numbers and back again be performed?
Even if such conversions should be kept to a minimum in order to maintain audio quality, they can hardly be avoided altogether. Therefore a clear understanding of the consequences of different alternatives is required.

*Is the performance of an audio application improved by eliminating denormalized floating point audio sample values by replacing them with the value zero?*

Not eliminating them is reported as a cause of possible high CPU load. These values are several hundred dB below floating point “0 dB full scale” and their elimination should therefore be totally without an audible effect. As an extreme precaution though, this inaudibility could be experimentally verified, but no reports on this were found.

*Should dither be applied or not in the context of floating point audio sample representation?*

There are theoretical treatments on the problems with floating point dither, but no simple solutions. Perceptual evaluations are presumably needed to resolve the issue, which would involve extremely demanding listening tests.

*If dither should be applied, how should it be implemented?*

A clear distinction has to be made between two different cases of requantization: 1) from a higher resolution floating point format to a lower resolution floating point format, and 2) from a higher resolution floating point format to a lower resolution fixed point integer format.

*Does the use of amplitude correlated floating point dither result in audible distortion?*

This type of dither is proposed in the literature, but no objective measurements or perceptual evaluations of its use were found.

*How should the audio quality of an audio subsystem, in particular the audio software signal path, be evaluated in a rational way?*

There are treatments on objective measurements on entire computer audio
systems in the literature, but no reports that include explicit considerations of the audio software signal path resulting from the use of a specific native audio API.

*What is the audio quality of each user and kernel mode software component in an audio software signal path?*

No reports on this were found in the literature whatsoever. It is still of great importance however in order to establish a clear understanding of the resulting overall audio quality of the audio subsystem as a whole.
6 Discussion

6.1 Requirements specification and testing

As already mentioned in section 2.1, a good requirements specification is of crucial importance for a successful software development project, and this is particularly true for audio software development. It must also once again be stressed that a considerable element of professional audio production knowledge should be involved in all phases of the audio software development process. From the treatment in previous chapters it is evident that quite different audio software signal paths result from different choices of native audio APIs on different operating system platforms.

A conclusion of this is that the requirements specification phase for audio software development must include deliberate control of the resulting audio software signal path through the audio subsystem, as a means to deterministically predict audio quality. Another difficulty lies in finding comprehensive audio quality test cases to evaluate that the audio application meets the stipulated requirements, since much of the audio software signal path demands black box, i.e. no source code, testing.

Methods for this type of work in an audio quality aware software development model are still missing from literature.

6.2 Limitations

A known limitation in this work is that no experimental audio quality evaluations were made. However, without a clear understanding of an audio subsystem on a very detailed software level, experiments, e.g. listening tests, risk to contain too many uncontrolled technical factors to provide reliable results. Therefore, the work in this essay was focused on contributing to the establishment of a clear understanding of the underlying software issues in studies of audio subsystems in computer audio systems.

The audio software signal paths discussed in this essay are constructed by user and kernel mode software components that are part of a specific operating system and its current version. Hence, they can be said to constitute moving targets, which could be another limitation from the perspective of academic research for the possible generalisability of the results. However, these moving targets do not move faster than that changes are introduced in intervals of several years’ time. Therefore, a solid understanding of audio quality aspects of an audio subsystem, defined by a specific operating system, can be established, and this knowledge then in turn constitutes the basis for studies of future audio subsystems in future operating system releases.

Without the systematic establishment of such knowledge, audio quality issues in computer audio systems remain largely intangible, and this poses
a problem for the use of e.g. computer audio system based high quality listening test audio applications.

6.3 Validity and reliability

Discussing validity and reliability in the context of the study conducted in this essay is not as clear-cut as perhaps could be claimed for experimentally oriented studies in well established research fields. Audio software development from an audio quality perspective is a new area of study within the subject of Audio Technology at Luleå University of Technology. Therefore a summarizing and integrating research review was needed to make an inventory of the state of the art, as a starting point for other studies within the field.

The validity and reliability in this study and its results must primarily be attributed to the discerning skills and acquired knowledge gained by the author through several years of experience as a Research Engineer at Swedish Radio, including international work within the European Broadcasting Union (EBU). This experience constituted the basis upon which the search for relevant literature and its further evaluation and analysis was built.

As for the resulting questions presented in chapter 5, their validity can be claimed from the observation that they still remain unanswered in the literature.

6.4 Conclusions

The audio software signal path through an audio subsystem in a computer audio system can be quite complex, and is not easy to unambiguously elucidate. Many software related issues must be carefully considered in an audio quality perspective when developing audio applications. The field is not well documented in the literature.

One reason for the lack of a widespread knowledge of audio software development from an audio quality perspective is possibly found in the following remark by Jones et al [11]:

“Much of the development in PC audio has been by designers whose background and training is in areas other than audio, …”

Another similar statement is by Ternström [26]:

“The personal computer has brought a revolution in enabling us to do lab work in acoustics and other science on the office desktop. In some ways, the cost of equipment is not nearly the problem that it was a decade or two ago. Still, we must bear in mind that today’s PC is a consumer device. As such, it is
designed to meet requirements that are often very different from those that apply for scientific work.”

In this “consumer device” perspective lies the challenge of using computers for professional audio production and research work.

6.5 Further work

An obvious improvement and direction of further work is to perform experimental work with software prototypes in order to further deepen, refine and complement some of the literature based findings presented in this essay.

As a means of gaining a better understanding of native audio API internals, a detailed study of ALSA on Linux would probably be a fruitful starting point, since both ALSA and Linux are open source software.

The questions in chapter 5 on conversions and dither in a floating point context could also be further addressed by software prototype experiments.

As an aid in the evaluation of the audio quality of an audio subsystem in a computer audio system a set of audio software quality metrics would be helpful. This is related to the discussion in 6.1 above, and probably represents a long-term perspective of future work.

An example of further work requiring quite a significant effort, would be the development of a software tool to automatically and unambiguously elucidate the audio software signal path through an audio subsystem, including internal audio sample formats of constituent user and kernel mode software components, and an associated level diagram. This is presumably not a trivial task though, even from a single-platform perspective, and would be even more complicated from a multi-platform perspective – Windows, Mac OS X and Linux.
References


