A Standard Audio API for C++: Motivation, Scope, and Basic Design

Guy Somberg (guy@gameaudioprogrammer.com)
Guy Davidson (guy@hatcat.com)
Timur Doumler (papers@timur.audio)

Document #: P1386R0
Date: 2019-01-21
Project: Programming Language C++
Audience: SG13

"C++ is there to deal with hardware at a low level, and to abstract away from it with zero overhead."

– Bjarne Stroustrup, Cpp.chat Episode #44[1]

Abstract

This paper proposes to add a low-level audio API to the C++ standard library. It allows a C++ program to interact with the machine's sound card, and provides basic data structures for processing audio data. We argue why such an API is important to have in the standard, why existing solutions are insufficient, and the scope and target audience we envision for it.

We provide a brief introduction into the basics of digitally representing and processing audio data, and introduce essential concepts such as audio devices, channels, frames, buffers, and samples.

We then sketch out a first design of such a low-level device access API, as well as examples how to use it. A first draft implementation of the API is available online.

Finally, we mention some open design questions that require feedback from the committee, and discuss additional features that are not yet part of the API as presented but will be added in future papers.

[1] See [CppChat]. The relevant quote is at approximately 55:20 in the video.

Contents

1 Motivation **3**
1.1 Why Does C++ Need Audio? 3
1.2 Audio as a Part of Human Computer Interaction 4
1.2 Why What We Have Today is Insufficient 4
1.3 Why not Boost.Audio? 6

2 Previous Work **7**

3 Scope **7**

4 Target Audience **8**

5 Background **8**
5.1 What is Audio? 8
5.2 Representing audio data in C++ 9
5.2.1 Samples 9
5.2.2 Frames and Channels 10
5.2.3 Buffers 10
5.3 Audio Devices 11
5.4 Audio Software Interfaces 11

6 Design **12**
6.1 Design Principles 12
6.2 Design Overview 12
6.3 Device Selection 13
6.4 Class device 13
6.5 Class buffer_list 16
6.6 Class buffer 16
6.7 Class buffer_view 17
6.8 Class strided_span 18
6.9 Algorithms 20

7 Example Usage **20**
7.1 White Noise 20
7.2 Process in the Main Thread 21
7.3 Sine Wave 22
7.4 Low-Latency Processing of Microphone Input 22

8 Reference Implementation **24**

9 Polls for the Committee **24**
9.1 General Polls 24
9.2 Underlying Type of a Buffer 24
9.3 Alternate Backends 25

10 Future steps **26**
10.1 Changing the Device Configuration 26
10.2 Detecting Device Configuration Changes 26
10.3 Combining Multiple Devices 27
10.4 Channel Convenience Naming 27
10.5 Clocks and Timestamps 27
10.6 Exclusive vs Shared Mode 27

11 References **28**

Document Revision History

R0, 2019-01-21: Initial version.

1 Motivation

1.1 Why Does C++ Need Audio?

Almost every computer, phone, and embedded device on the market today comes with some form of audio output and (in many cases) input, yet there is no out-of-the-box support for audio in the C++ language. Developers have to use a multitude of different platform-specific APIs or middleware cross-platform frameworks. Wouldn't it be great if the basic functionality of talking to your sound card would come for free with every C++ compiler, as part of the C++ standard library? Not only would it allow one to write truly cross-platform audio code, but it would also lower the barrier of entry for learning.

The principles of pulse-code modulation (PCM) go back to at least the 1920s [PCM], and commodity PC hardware has been doing this exact thing since at least the early 1990s [SoundCard] if not earlier. That gives us between 30 and 100 years of experience doing audio the same way. These fundamentals have withstood the test of time - they have remained virtually unchanged from its humble beginnings, through the invention of the MP3, surround sound, and even now into the heyday of ambisonics and virtual reality. Throughout all of this time, PCM is the *lingua franca* of audio. It is time to provide an interface to audio devices in the C++ standard.

The job of the standard is to standardise existing practice. This proposal intends to draw together current practice and present a way forward for a standard audio interface designed using modern C++ features.

1.2 Audio as a Part of Human Computer Interaction

Since the advent of C, computing has changed considerably with the introduction and widespread availability of graphics and the desktop metaphor. Human Computer Interaction (HCI) is the field of study which considers how people interact with computers and technology, and has expanded greatly since the introduction of personal computing. C++ is a systems programming language widely used on the server as well as the client (mobile phones and desktop computers). It is also a language which lacks library support for many fundamental aspects of client computing. If C++ is to be a language for the client as well as the server, it needs to complete its HCI support.

Games are often used to demonstrate the scope of requirements for HCI support. In order to implement even the simplest of primitive games, you need at a minimum the following fundamental tools:

- A canvas to draw on.
- Graphics support to draw specific figures.
- Input support for receiving user control events.
- Audio support for providing additional reinforcing feedback.

Currently, the C++ standard library provides none of these tools: it is impossible for a C++ programmer to create even a rudimentary interactive application with the tools built into the box. She must reach for one or more third-party libraries to implement all of these features. Either she must research the APIs offered by her program's various supported host systems to access these features, or she must find a separate library that abstracts the platform away. In any case, these APIs will potentially change from host to host or library to library, requiring her to learn each library's particular individual quirks.

If C++ is there to deal with the hardware at a low level, as Bjarne Stroustrup said, then we must have access to all of the hardware, and that includes the audio hardware.

Audio playback and recording has been solved many times over - a large number of both proprietary and open-source libraries have been developed and implemented on a myriad of platforms in an attempt to provide a universal API. Examples libraries include Wwise, OpenAL, Miles Sound System, Web Audio, PortAudio, RtAudio, JUCE, and FMOD to name a few. [AudioLibs] lists 38 libraries at the time of writing. While some of these APIs implement higher-level abstractions such as DSP graphs or fancy tooling, at a fundamental level they are all doing the exact same thing in the exact same way.

1.2 Why What We Have Today is Insufficient

The corpus of audio libraries as it exists today has a few fundamental problems:

- The libraries are often platform-specific.
- There is a lot of boilerplate code that cannot be shared among platforms.
- The libraries are not written to be able to take advantage of modern syntax and semantics.

Consider the “Hello World” of audio programming: the playback of white noise, which is generated by sending random sample data to the output device. Let’s examine what this code will look like on MacOS with the CoreAudio API and on Windows with WASAPI, the foundational audio libraries on their respective platforms. (The code in this section has been shrunk to illegibility on purpose in order to show the sheer amount of boilerplate required.)

MacOS CoreAudio

```
AudioObjectPropertyAddress pa = {
  kAudioHardwarePropertyDefaultInputDevice,
  kAudioObjectPropertyScopeGlobal,
  kAudioObjectPropertyElementMaster
};
uint32_t dataSize;
if (noErr != AudioObjectGetPropertyDataSize(
    kAudioObjectSystemObject, &pa, 0, nullptr, &dataSize)
    || dataSize != sizeof(AudioDeviceID))
  return err;

AudioDeviceID deviceID;
if (noErr != AudioObjectGetPropertyData(
    kAudioObjectSystemObject, &pa, 0, nullptr, &dataSize,
&deviceID))
  return err;

AudioDeviceIOProcID ioProcID;
if (noErr != AudioDeviceCreateIOProcID(
    deviceID, ioProc, nullptr, &ioProcID))
  return err;
if (noErr != AudioDeviceStart(deviceID, ioProc)) {
  AudioDeviceDestroyIOProcID(deviceID, ioProcID);
  return err;
}

AudioDeviceStop(deviceID, ioProc);
AudioDeviceDestroyIOProcID(deviceID, ioProcID);

OSStatus ioProc(AudioObjectID deviceID,
               const AudioTimeStamp*,
               const AudioBufferList*,
               const AudioTimeStamp*,
               AudioBufferList* outputData,
               const AudioTimeStamp*,
               void*)
{
  if (outputData != nullptr) {
    const size_t numBuffers = outputData->mNumberBuffers;

    for (size_t iBuffer = 0; iBuffer < numBuffers; ++iBuffer) {
      const AudioBuffer& buffer = outputData->mBuffers[iBuffer];
      const size_t numSamples = buffer.mDataByteSize /
sizeof(float);

      float* pDataFloat = reinterpret_cast<float*>(buffer.mData);
      for (size_t i = 0; i < buffer.mDataByteSize; ++i) {
        pDataFloat[i] = get_random_sample_value();
      }
    }
  }
  return noErr;
}
```

Windows WASAPI

```
CoCreateInstance(CLSID_MMDeviceEnumerator, NULL, CLSCTX_ALL,
    IID_IMMDeviceEnumerator, (void**)&pEnumerator);
pEnumerator->GetDefaultAudioEndpoint(eRender, eConsole,
&pDevice);
pDevice->Activate(IID_IAudioClient, CLSCTX_ALL, NULL,
    (void**)&pAudioClient);

pAudioClient->GetMixFormat(&pwfx);
pAudioClient->Initialize(AUDCLNT_SHAREMODE_SHARED, 0,
    hnsRequestedDuration, 0, pwfx, NULL);
pAudioClient->GetBufferSize(&bufferFrameCount);
pAudioClient->GetService(IID_IAudioRenderClient,
(void**)&pRenderClient);

pMySource->SetFormat(pwfx);

pRenderClient->GetBuffer(bufferFrameCount, &pData);
pRenderClient->ReleaseBuffer(bufferFrameCount, flags);

hnsActualDuration =
    (double)REFTIMES_PER_SEC * bufferFrameCount /
pwfx->nSamplesPerSec;
pAudioClient->Start();

while (flags != AUDCLNT_BUFFERFLAGS_SILENT) {
  Sleep((DWORD)(hnsActualDuration/REFTIMES_PER_MILLISEC/2));
  pAudioClient->GetCurrentPadding(&numFramesPadding);
  numFramesAvailable = bufferFrameCount - numFramesPadding;
  pRenderClient->GetBuffer(numFramesAvailable, &pData);

  float* pDataFloat = reinterpret_cast<float*>(pData);
  for(int i=0; i<numFramesAvailable; i++) {
    pDataFloat[i] = get_random_sample_value();
  }

  pRenderClient->ReleaseBuffer(numFramesAvailable, flags);
}

Sleep((DWORD)(hnsActualDuration/REFTIMES_PER_MILLISEC/2));
pAudioClient->Stop();
```

In both of these examples, the large majority of this code simply sets up devices and buffers: only the sample-generating code (in blue) actually updates the output buffer in any way. Note that the sample-generating code is basically identical between the CoreAudio and WASAPI code[2].

The amount of boilerplate code has prompted the creation of several libraries which attempt to abstract devices and buffers. The same example using JUCE looks like this:

[2] If you can read it at that font size...

JUCE

```
class MainComponent : public AudioAppComponent
{
public:
  MainComponent() {
    setAudioChannels (2, 2);
  }

  ~MainComponent() {
    shutdownAudio();
  }

  void prepareToPlay (int samplesPerBlockExpected, double sampleRate) override {}

  void getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill) override {
    for (auto channel = 0; channel < bufferToFill.buffer->getNumChannels(); ++channel) {
      auto *buffer = bufferToFill.buffer->getWritePointer(channel, bufferToFill.startSample);
      for (auto sample = 0; sample < bufferToFill.numSamples; ++sample)
        buffer[sample] = get_random_sample_value();
    }
  }

  void releaseResources() override {}
  void paint (Graphics&) override {}
  void resized() override {}

  JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainComponent)
};
```

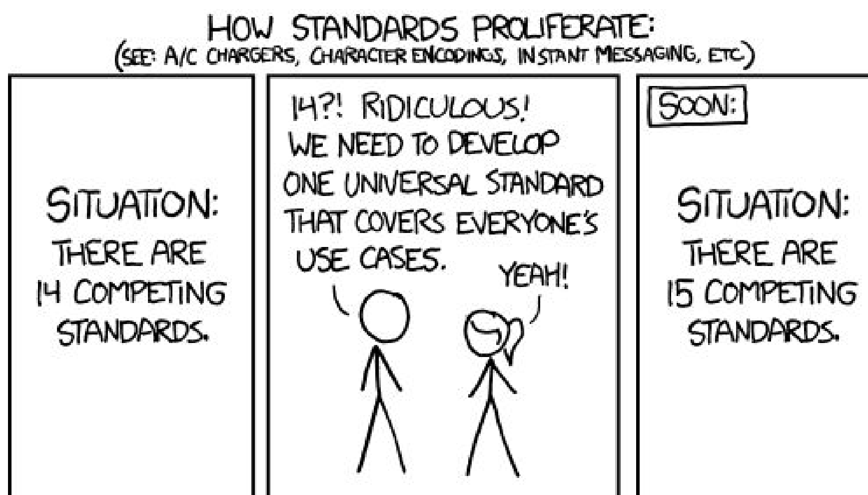
In this case, the abstraction is achieved by declaring a base class and requiring the client to override several member functions. While this does require less code, there is still redundancy in the form of four empty overridden functions, as well as a macro hiding a chunk of housekeeping. And, once again, our buffer-filling (blue) code is nearly identical.

In the first two code samples (CoreAudio and WASAPI), the code is calling a C API, and thus is limited in its ability to take advantage of modern C++ abstractions. The third code sample (JUCE) could have been written using C++98, with only the override keyword hinting at modern implementation.

New C++ language features have made it possible to write clearer, more concise code. It is the authors' intent to specify a modern interface to audio programming.

1.3 Why not Boost.Audio?

Obligatory reference to xkcd:



Source: https://xkcd.com/927/

Consider that a hypothetical Boost.Audio library existed with some interface not dissimilar to what is described in this paper. This is great for everybody who has access to a platform that Boost supports, and who is on a team that includes the boost libraries. But, let us say that a new embedded chip or a new operating system comes on the market with a new and different API for communicating with the audio hardware. How do users of a hypothetical Boost.Audio library access the audio hardware?

The problem is that Boost.Audio does not support this new audio hardware out of the box. Users of these new systems must either modify Boost.Audio to support their hardware (hopefully submitting a patch to Boost in the process), or they have to abandon their use of Boost.Audio for the unsupported platforms and split their codebase.

If, on the other hand, audio capabilities were a part of the standard library provided by the compiler, then the embedded chip or OS vendor could write one single high-quality implementation, ship it with the standard library, and all of their customers will immediately be able to use their existing `std::audio` code on the new hardware.

2 Previous Work

This is the first paper formally proposing audio functionality for the C++ standard. To our knowledge, the most serious discussion so far of such an idea is a thread from 2016 on the Future C++ Proposals Google group (see [Forum]). The author was initially sketching out a higher-level approach to audio based on stream-like semantics. Some of the criticism in the ensuing discussion was that this wasn't sufficiently low-level, didn't give direct access to the actual underlying audio data (samples, frames) and basic operations on it such as deinterleaving, wasn't directly based on existing audio libraries, and didn't have a way to interact with the concrete audio device used by the OS for audio I/O. The proposal we present here fulfils all of those requirements.

Later parts of the Google group discussion explore different design ideas towards a more universal low-level API for audio, some of which are part of our design as well.

Before publishing this paper, we presented an earlier draft of our proposal in November 2018 at the Audio Developer Conference in London (see [ADC]). We received overwhelmingly positive feedback for this proposal from the industry. We also received lots of useful technical feedback on the API design, and incorporated it into this paper.

3 Scope

This paper describes a low-level audio playback and recording device access API, which will function for a large class of audio devices. This API will work on commodity PC hardware (Windows, MacOS, Linux, etc.), microcontrollers, phones and tablets, big iron, exotic

hardware configurations, and even devices with no audio hardware at all[3]. Additionally, we target non-realtime use cases for command-line or GUI tools that want to operate on audio data without actually rendering it to an audio device.

Defining what is not supported is as much important as defining what is. This paper does not seek to implement any features related to MIDI, FM synthesis, audio file parsing/decompression/playback, buffered streaming from disk or network, or a DSP graph[4]. At least some of these omitted features are in scope for a `std::audio` library, but those features will come either in later revisions of this paper, or in papers of their own once the fundamentals have been laid out in this document.

4 Target Audience

Because the API presented in this paper is a low-level audio device access API, it is targeted at two broad categories of people: audio professionals and people learning low-level audio programming. The classes and algorithms are the same algorithms that professionals currently have to write for their target platforms, and they are designed for minimum overhead. At the same time, a beginner who wants to learn low-level audio programming will find the interfaces intuitive and expressive because they map directly to the fundamental concepts of audio.

However, because this API does not (yet) provide any facilities for functionality like loading and playing audio files, there is a category of programmers for whom this library is too low-level. For those people, we hope to include a suite of libraries in succession papers once this paper has been put through its paces.

5 Background

5.1 What is Audio?

Fundamentally, audio can be modeled as waves in an elastic medium. In our normal everyday experience, the elastic medium is air, and the waves are air pressure waves. The differences in air pressure is sensed by our ears, and our brains interpret the signals as sound. For more details on the fundamentals of audio and hearing, along with additional references, see Chapter 1 by Stephen McCaul of [GAP]. Additionally, for an interactive document which can help to create an intuition for sound and audio, see [Waveforms].

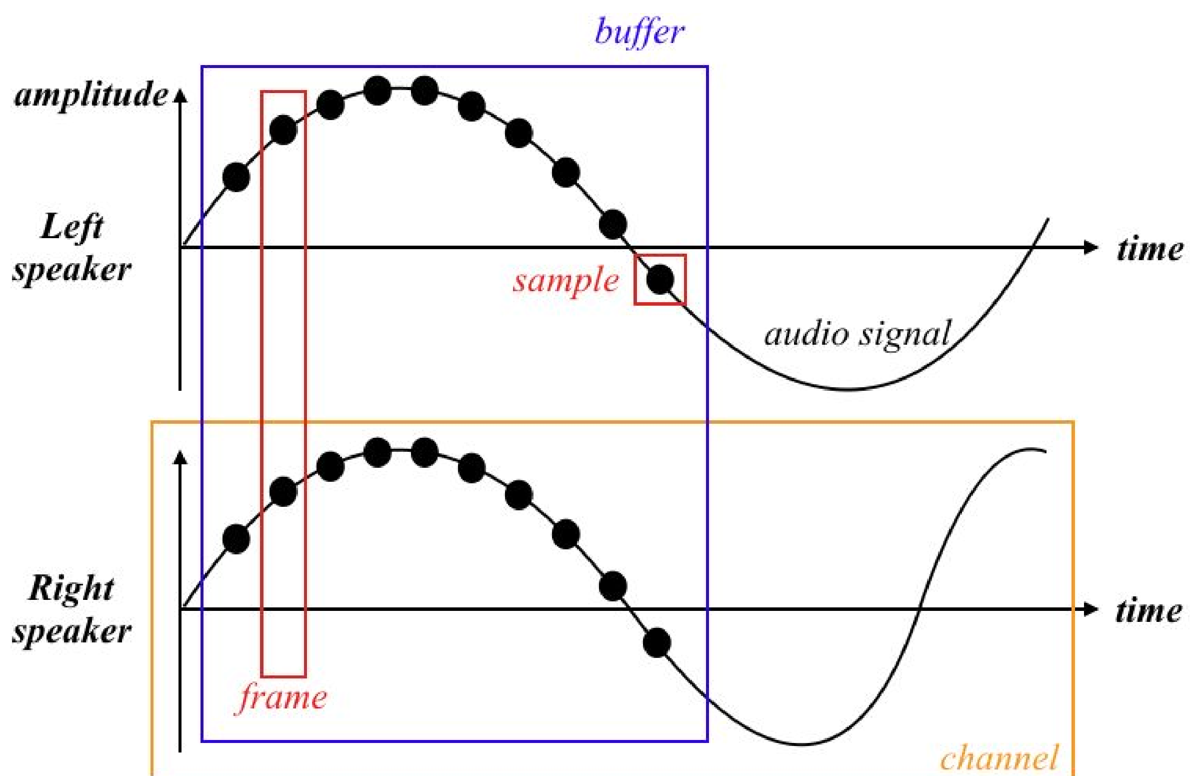
From chapter 1 of [GAP]:
"Any one-dimensional physical property can be used to represent air pressure at an instant in time. If that property can change over time, it can represent audio. Common examples

[3] Obviously, devices with no audio hardware will not generate any audio, but the API is aware of such devices and respects the principle that "you don't pay for what you don't use" on such devices.
[4] DSP graphs are the most common way to represent complex audio processing chains.

are voltage (the most common analog), current (dynamic microphones), optical transitivity (film), magnetic orientation (magnetic tape), and physical displacement (records).
[...]
The ubiquitous representation for processing sound [...] is pulse-code modulation, or PCM. This representation consists of a sequence of [values] that represent the sound pressure at equally spaced times."

More precisely, the representation most commonly used is Linear PCM (LPCM). There are other digital representations such as delta modulation, but they are much less widely-used. In this paper, we have deliberately chosen LPCM because it is the de facto standard for audio applications and drivers.

5.2 Representing audio data in C++



5.2.1 Samples

A *sample* is the fundamental unit of audio, which represents the amplitude of an audio signal at a particular point in time. It is represented as a single numeric value, typically either a floating point number in the range -1..+1 or a signed integer, but may also be another type that is appropriate to the target platform. The *sample rate* or sampling frequency is the number of samples per second. Typical sample rates are 44,100 Hz, 48,000 Hz, and 96,000 Hz. The *bit depth* is the number of bits of information in each sample, which can be lower than the number of bits available in the numeric data type used. Typical bit depths are 16 bit or 24 bit.

5.2.2 Frames and Channels

A *frame* is a collection of samples referring to the same point in time, one for each output (typically a speaker) or input (typically a microphone). For example, a stereo (two-speaker) output will have 2 samples in the frame, one for the left speaker and one for the right. A "5.1" channel surround sound system will have six samples in the frame: left, center, right, surround left, surround right, and LFE (low-frequency emitter, typically referred to as a "subwoofer"). Each sample within a frame is targeted at a particular speaker, and we refer to the collection of samples targeted for a particular speaker as a *channel*.

5.2.3 Buffers

A *buffer* is a collection of frames, typically laid out in a contiguous array in memory. Using such a buffer for exchanging data with the sound card greatly reduces the communication overhead compared to exchanging individual samples and frames, and is therefore the established method. On the other hand, buffers increase the latency of such data exchange. The tradeoff between performance and latency can be controlled with the *buffer size*. This will often be a power-of-two number. On desktop machines and phones, typical buffer sizes are between 64 and 1024 samples per buffer.

There are two orderings for buffers: interleaved and deinterleaved. In an interleaved buffer, the channels of each frame are laid out sequentially, followed by the next frame. In a deinterleaved buffer, the channels of each frame are laid out sequentially, followed by the next channel laid out sequentially.

It is probably easiest to view this visually. In the following tables, each square represents a single sample, L stands for "left channel", and R stands for "right channel". Each buffer contains four frames of stereo audio data.

Interleaved:

L	R	L	R	L	R	L	R

Deinterleaved:

L	L	L	L	R	R	R	R

Another example, this time with a 5.1-channel setup. Here L = left, R = right, C = center, SL = surround left, SR = surround right, LFE = low-frequency emitter. In order to fit onto a page, there are only three frames in these buffers.

Interleaved:

L	R	C	SL	SR	LFE	L	R	C	SL	SR	LFE	L	R	C	SL	SR	LFE

Deinterleaved:

L	L	L	R	R	R	C	C	C	SL	SL	SL	SR	SR	SR	LFE	LFE	LFE

5.3 Audio Devices

A *device* is the software interface to the physical hardware that is outputting or inputting audio buffers. Audio hardware consists of a digital to analog converter (DAC) for output devices (speakers), and/or an analog to digital converter (ADC) for input devices (microphones).

Under the hood, devices have a double buffer of audio data[5]: the device consumes one buffer while the software fills the other. However, the audio device ***does not wait*** for the previous buffer to complete before swapping the buffers. In fact, it cannot, because time moves forward whether or not we are ready for it. This is important, because it means that audio is a near real-time system[6]. Missing even a single sample is unacceptable in audio code, and will typically result in an audible glitch.

Every time a buffer context switch occurs, the device signals that its memory buffer is ready to be filled (output) or consumed (input). At that moment, the program has a very short window of time - typically a few milliseconds at most[7] - in which to fill or consume the buffer with audio data before the buffer is switched again. At a fundamental level, the design proposed in this document is about hooking into this moment in time when the buffer is ready and exposing it to user code.

5.4 Audio Software Interfaces

There are two broad categories of audio APIs: polling APIs and callback APIs.

In a callback API, the operating system fabricates a thread on your program's behalf and calls into user code through that thread every time a buffer becomes available. One example of a callback-based API is MacOS CoreAudio.

In a polling API, the user code must check periodically whether a buffer is available. The polling function will return a valid buffer when one is available, and the user code can process it. Some polling systems will provide an event handle that can be used to block the user code until a buffer is available. Examples of polling APIs are Windows WASAPI (which includes an event handle) and any embedded devices which do not have an operating system (and therefore cannot create threads).

---

[5] As with everything in C++, this explanation follows the "as if" rule. The actual mechanics are more complex, but the distinctions are not important to this paper.
[6] Audio processing exhibits many of the properties of a hard real-time system, but in most contexts there is an operating system and a thread scheduler between the audio software and the chip. So, although it is tempting to call it so, "hard real-time" is not technically correct in most cases.
[7] An example: a common sample rate is 48,000 Hz, and a common buffer size is 128 samples. 128 samples divided by 48,000 samples per second gives just 2.66 milliseconds to fill the buffer.

On systems where threads are available, the audio will typically be driven by a thread (in callback APIs, there is no other choice!). Due to the time restrictions, it is critically important that these threads perform no operations that could possibly block for an indeterminate length of time. This means that there can be no file or network I/O, no locks, and no memory allocations. Algorithms and data structures that are used in the audio thread are typically either lock-free or wait-free. Additionally, the thread is typically run at the highest available OS priority.

6 Design

6.1 Design Principles

The API in its current form is a low-level device access API, intended to be the least-common denominator. As the quote from the first page of this document indicates, the API must abstract the hardware at a low level, and leave no room for a lower-level interface. Thus, this API, by necessity, is bare-bones. However, it lays the foundations for higher-level abstractions that can come in the future, or which users can build upon to create their own software and libraries.

Another thing to note is that this paper is not inventing anything. While the API surface of this library is distinctive, it is an attempt to standardize a common denominator among several popular low-level device access APIs

6.2 Design Overview

This paper (the R0 version) will cover a high-level design of the various classes and structures and their relationships. If we get positive feedback on this design, then future versions will include proposed wording and completely fleshed-out APIs.

Broadly, the proposed design provides abstractions for each of the concepts defined in Section 5 of this document:

- **`std::experimental::audio::device_list`** - A list of devices that are available on the system.
- **`std::experimental::audio::device`** - Communicates with an audio input or output driver.
- **`std::experimental::audio::buffer_list`** - Passed by a device to the audio callback. Provides zero or more input and output buffers that the device is expecting to be filled (for output devices) or containing audio data (for input devices).
- **`std::experimental::audio::buffer`** - A range of multi-channel audio data.
- **`std::experimental::audio::buffer_view`** - A view into a buffer that interprets it as a collection of channels or a collection of frames.
- **`std::experimental::audio::strided_span`** - A view into a buffer that has a known length and has a “stride” into the underlying data such that all of the channels of a single frame, or all of the individual channels of the buffer are iterated over.

The remainder of this section will go into greater detail about the specific classes, their usage, and their APIs. Unless stated otherwise, all free functions and classes are in the `std::experimental::audio` namespace.

Note that this API is intentionally incomplete so that the signal does not get overwhelmed by the noise. For example, we have left out constructors, destructors, allocator support, and thorough noexcept to name but a few items. These functions and more will come in future revisions of this document, once we have encouragement from SG13 to continue in this direction.

6.3 Device Selection

The first entry point to `std::audio` is to figure out which device you wish to communicate with using the device selection routines.

```
device get_default_input_device()
device get_default_output_device()
```

Returns: device objects referring to the system-default input or output device. If there is no default input or output device, returns a default-constructed device.

```
device_list get_input_device_list()
device_list get_output_device_list()
```

Returns: device_list objects which will iterate over the input or output devices that are currently available on the system.

```
class device_list {
  iterator begin();
  iterator end();
};
```

6.4 Class `device`

After using the device selection API, you will end up with a `device` object. The `device` class communicates with the underlying audio driver, and can be run in a threaded mode or a polling mode. A device can have only inputs, only outputs, or both inputs and outputs.

```
enum class buffer_order
{
  interleaved,
  deinterleaved
};

class device
{
  using sample_type = <tbd>
  using callback = void(*)(device&, buffer_list&);
```

```
  buffer_order get_native_ordering() const noexcept;
  size_t get_sample_rate() const noexcept;
  size_t get_buffer_size() const noexcept;
  size_t get_bit_depth() const noexcept;

  void connect(callback cb);

  void start();
  void stop();

  void wait() const;
  template<class Rep, class Period>
  void wait_for(std::chrono::duration<Rep, Period> rel_time) const;
  template<class Clock, class Duration>
  void wait_until(std::chrono::time_point<Clock, Duration> abs_time)
const;

  void process(callback cb);
};
```

`buffer_order device::get_native_ordering() const noexcept`
Returns: The buffer order of the data in the buffers contained in the `buffer_list` passed to the callback, which is accessible via `buffer::raw()`.

`size_t device::get_sample_rate() const noexcept`
Returns: The sample rate that the device is configured to play at.

`size_t device::get_buffer_size() const noexcept`
Returns: The size of the buffer in bytes that the device will provide to the callback.

`size_t device::get_bit_depth() const noexcept`
Returns: The number of bits actually used in the data type. This number is guaranteed to be less than or equal to `sizeof(sample_type) * CHAR_BIT`.

`void device::connect(callback cb)`
Results: Attaches a callback to execute when the audio device is ready to receive one or more buffers of audio data (output devices) or has generated one or more buffers of audio data (input devices). If this function is being used, it must be called before calling `start()`. If this function is called after `start()`, an exception is thrown. The thread that is generated acts as if it contained the following code:

```
void thread_func() {
  while(!stopped) {
    wait();
    process(cb);
```

```
  }
}
```

`void start()`
Results: Initializes the underlying audio device to begin requesting (output devices) or generating (input devices) audio data. If `connect()` has been called on this device, the audio device will be driven by a separate thread, which is started at this point. The thread may be created by the library or by the operating system. If a thread has been requested but cannot be created, then an exception is thrown.

`void stop()`
Results: Shuts down the underlying audio device so that it no longer requests (output devices) or generates (input devices) audio data. If a thread was created, it is joined or detached when this function returns. If `start()` has not been called, then this function has no effect.

`void device::wait() const`
Results: Waits until the device is ready with at least one buffer of audio data. If the device is configured to drive a thread (by calling `connect()`), or the device is not currently running (`start()` has not yet been called or `stop()` has been called), then this function returns immediately.

```
template<class Rep, class Period>
void wait_for(std::chrono::duration<Rep, Period> rel_time) const
```
Results: Waits until either the device is ready with at least one buffer of audio data, or the specific duration has elapsed. If the device is configured to drive a thread (by calling `connect()`), or the device is not currently running (`start()` has not yet been called or `stop()` has been called), then this function returns immediately.

```
template<class Clock, class Duration>
void wait_until(std::chrono::time_point<Clock, Duration> abs_time) const
```
Results: Waits until either the device is ready with at least one buffer of audio data, or until the absolute timeout has expired. If the device is configured to drive a thread (by calling `connect()`), or the device is not currently running (`start()` has not yet been called or `stop()` has been called), then this function returns immediately.

`void process(callback cb);`
Results: Checks to see if the device has at least one buffer of audio data available. If so, it calls the callback with `*this` and a `buffer_list` referencing all available buffers. If the device is configured to drive a thread (by calling `connect()`), the device is not currently running (`start()` has not yet been called or `stop()` has been called), or there are no audio buffers available, then this function returns immediately.

6.5 Class `buffer_list`

A `buffer_list` is a wrapper for two collections: one for input buffers and one for output buffers. Most consumer devices (in fact, all systems that these authors know about) will have either zero or one entry in the collection - a fact which implementers are expected to take advantage of. However, this API does not wish to preclude devices which are unknown to the authors - or which do not yet exist - which may provide multiple buffers to be filled.

```
class buffer_list
{
  span<buffer> input_buffers() const noexcept;
  span<buffer> output_buffers() const noexcept;
};
```

`span<buffer> buffer_list::input_buffers() const noexcept`
Returns: A `span` containing zero or more buffers of audio data that has been generated by the device.

`span<buffer> buffer_list::output_buffers() const noexcept`
Returns: A `span` containing zero or more buffers of audio data that the device has requested to be filled.

6.6 Class `buffer`

A `buffer` is a random access collection of samples with different accessors for raw, interlaced, and deinterlaced views of the data.

```
class buffer
{
  buffer_order get_ordering() const noexcept;

  span<value_type> raw() const noexcept;

  buffer_view channels() const noexcept;
  buffer_view frames() const noexcept;
};
```

`buffer_order buffer::get_ordering() const noexcept`
Returns: The buffer order of the underlying buffer, as returned by `raw()`.

`span<value_type> buffer::raw() const noexcept`
Returns: The underlying buffer.

`buffer_view buffer::channels() const noexcept`

Returns: A view of the underlying buffer as a deinterleaved collection of collections of channels.

```
buffer_view buffer::frames() const noexcept
```

Returns: A view of the underlying buffer as an interleaved collection of collections of frames.

6.7 Class `buffer_view`

A `buffer_view` is a read-only collection view of a `buffer` that is crafted such that it iterates over the channels or frames of the underlying buffer as appropriate. Note that it is not actually a container because it returns by value rather than by reference. The type that it returns is itself a collection view into the underlying buffer, which means that it does not own the data, so returning a reference is inappropriate. This differs from the normal container requirements in that it is not actually storing its own data, but rather is a view into some data that is owned by the buffer.

```
class buffer_view
{
  using value_type = strided_span;
  using size_type = size_t;
  // other types as necessary...

  size_type size() const noexcept;
  value_type operator[](size_t index) const noexcept;
  value_type operator[](size_t index) noexcept;

  value_type at() const;
  value_type at();

  class iterator;
  class const_iterator;

  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;

  iterator end();
  const_iterator end() const;
  const_iterator end() const;
  // other read-only container functions...
};
```

```
value_type operator[](size_t index) const
value_type operator[](size_t index)
```

Returns: A `strided_span` starting at the given index and with the appropriate stride to access the underlying buffer's data appropriately. For example, if the underlying data is

2-channel interleaved and this `buffer_view` was created from the `buffer::frames()` function, then `operator[]` will return a `strided_span` with a size of 2 and a stride of 1. If the underlying data is 4-channel deinterleaved and this `buffer_view` was created from the `buffer::channels()` function, then `operator[]` will return a `strided_span` with a size of 4 and a stride of `(buffer.raw().size() / 4)`.

6.8 Class `strided_span`

A `strided_span` is a span which supports skipping entries from the underlying buffer according to some stride. As with `buffer_view`, it is intended to access data owned by the `buffer`.

The purpose of `strided_span` is to step through the contents of a memory buffer owned by another entity in a specific order. Let us take one of the examples from section 5.2.3, reproduced here in its original form:

Interleaved:

L	R	C	SL	SR	LFE	L	R	C	SL	SR	LFE	L	R	C	SL	SR	LFE

Deinterleaved:

L	L	L	R	R	R	C	C	C	SL	SL	SL	SR	SR	SR	LFE	LFE	LFE

Let us say that we wish to examine this data as a sequence of frames, each one containing a Left, Right, Center, Surround Left, Surround Right, and LFE channel. We can implement our `strided_span` as an offset into the buffer combined with a stride and a total count:

```
template<class T>
class strided_span
{
  // ...
private:
  T* buffer;
  size_t offset;
  size_t stride;
  size_t total_count;
};
```

So, for this buffer, we will have the following:

	Interleaved	Deinterleaved
Frame 0	Offset: 0 Stride: 1 Count: 6	Offset: 0 Stride: 3 Count: 6

Frame 1	Offset: 6 Stride: 1 Count: 6	Offset: 1 Stride: 3 Count: 6
Frame 2	Offset: 12 Stride: 1 Count: 6	Offset: 2 Stride: 3 Count: 6

Matching the colors to the buffers, we get this as expected:

Interleaved:

L	R	C	SL	SR	LFE	L	R	C	SL	SR	LFE	L	R	C	SL	SR	LFE

Deinterleaved:

L	L	L	R	R	R	C	C	C	SL	SL	SL	SR	SR	SR	LFE	LFE	LFE

We can also use the strided_span to view the buffer's contents as an array of channels. This time we will use our two-channel stereo setup from section 5.2.3 as an example[8]:

Interleaved:

L	R	L	R	L	R	L	R

Deinterleaved:

L	L	L	L	R	R	R	R

Our table now looks like this:

	Interleaved	Deinterleaved
Left channel	Offset: 0 Stride: 2 Count: 4	Offset: 0 Stride: 1 Count: 4
Right channel	Offset: 1 Stride: 2 Count: 4	Offset: 4 Stride: 1 Count: 4

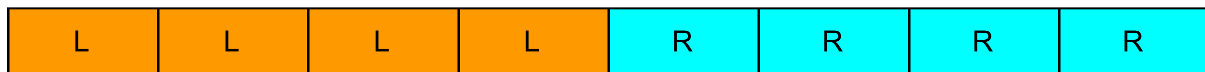
And, mapping the colors we get:

Interleaved:

L	R	L	R	L	R	L	R

[8] Mostly this is so that I don't have to come up with six different background colors to use.

Deinterleaved:



Note that in both of these examples, the stride and count are fixed, whereas the offset increments either by 1 or by the count.

6.9 Algorithms

Currently, the only algorithms are related to interleaving and deinterleaving of buffers. According to [Jain] there exists an O(n) time O(1) space algorithm for a two-way interleave, and the same paper posits that there exist such algorithms for k-way shuffles.

```
template<class RandomAccessIterator>
void interleave(
  RandomAccessIterator begin, RandomAccessIterator end, size_t channels)
```

Result: Assumes that the input range is deinterleaved. Swaps the members such that the input range is interleaved.

```
template<class RandomAccessIterator>
void deinterleave(
  RandomAccessIterator begin, RandomAccessIterator end, size_t channels)
```

Result: Assumes that the input range is interleaved. Swaps the members such that the input range is deinterleaved.

```
template<class RandomAccessIterator>
void set_buffer_order(
  RandomAccessIterator begin, RandomAccessIterator end, size_t channels,
  buffer_order input_order, buffer_order output_order)
```

Result: As-if the following code:

```
if(input_order == output_order)
      return;
if(output_ordering == buffer_ordering::interleaved)
      interleave(begin, end, channels);
else
      deinterleave(begin, end, channels);
```

7 Example Usage

7.1 White Noise

In audio, white noise refers to a random signal that has equal intensity at different frequencies [Noise]. We can generate white noise by picking uniformly distributed random

values within the allowed value range. We use the `minstd_rand` engine because it is smaller and faster than other generators.

```
random_device rd;
minstd_rand engine{rd()};
uniform_real_distribution<float> distribution{-1.0f, 1.0f};

float get_random_sample_value() {
  return distribution(engine);
}

int main() {
  using namespace std::experimental::audio;
  auto device = get_default_output_device();
  device.connect([](device& d, buffer_list& bl) {
    for (auto& buffer : bl.output_buffers()) {
      for (auto& channel : buffer.channels()) {
        for (auto& sample : channel) {
          sample = get_random_sample_value();
        }
      }
    }
  });

  device.start();

  while(true); // Spin forever
}
```

7.2 Process in the Main Thread

On systems with a polling API such as WASAPI on Windows, you can drive the audio from the main thread if you so choose.

```
int main() {
  using namespace std::experimental::audio;
  auto device = get_default_output_device();
  auto callback = [](device& d, buffer_list& bl) { /* ... */ };

  device.start();

  while(true) {
    device.wait();
    device.process(callback);
  }
}
```

7.3 Sine Wave

White noise is all well and good, but what if we want to actually generate something real? We'll generate a 440 Hz sine wave. (For musicians[9], that's an A.)

```
int main() {
  using namespace std::experimental::audio;

  auto d = get_default_output_device();

  const double frequency_hz = 440.0;
  const double delta = 2.0 * M_PI * frequency_hz / d.get_sample_rate();
  double phase = 0;

  d.connect([=](device&, buffer_list& bl) mutable {
    for (auto& buffer : bl.output_buffers()) {
      for (auto& frame : buffer.samples()) {
        auto next_sample =
          static_cast<device::sample_type>(std::sin(phase));
        phase += delta;
        for (auto& sample : frame)
          sample = next_sample;
      }
    }
  });

  device.start();

  while(true); // Spin forever
}
```

7.4 Low-Latency Processing of Microphone Input

Here we are reading input samples from the microphone, passing them to some library that performs processing on the raw memory buffer (this library assumes that the buffer is interleaved), and then writing the processed buffer to the device's output buffer. Note that, although this example is long compared with the others, it is still less than 30 lines of code (not including comments or whitespace) for a fairly complex task.

```
int main() {
  using namespace std;
  using namespace std::experimental::audio;
```

[9] Although some orchestras will tune to 441 Hz, or even 442 Hz. Interesting historical tidbit: the audio CD format runs at 44,100 Hz, which is exactly enough for 100 samples per pulse for an A at a 441 Hz.

```
// We will presume that this is an input/output device
auto d = get_default_output_device();

vector<device::sample_type> processed_data;
d.connect([&](device& d, buffer_list& bl) {
  // First process the input buffers
  for (auto& buffer : bl.input_buffers()) {
    // Get the raw input buffer and make sure that it is
    // in the correct order for the third-party library.
    auto raw_buffer = buffer.raw();
    set_buffer_order(begin(raw_buffer), end(raw_buffer),
                     buffer.channels().size(),
                     buffer.get_ordering(),
                     buffer_order::interleaved);

    // Pass the buffer to the third-party library to process
    ThirdPartyLib::Process(raw_buffer);

    // Copy the contents of the buffer to send to the output.
    size_t current_size = processed_data.size();
    size_t raw_buffer_size = raw_buffer.size();
    processed_data.resize(current_size + raw_buffer_size);
    copy(begin(raw_buffer), end(raw_buffer),
      processed_data.data() + current_size);
  }

  // Now process the output buffers
  for (auto& buffer : bl.output_buffers()) {
    // We are once again operating on raw buffers.
    auto raw_buffer = buffer.raw();

    // Grab the next processed buffer and make sure that it is
    // in the correct order
    span<device::sample_type> processed_samples
      { processed_data.data(),
        min(processed_data.size(), raw_buffer.size()) };

    // Put the processed data back into device native ordering.
    // We are assuming that input channel count is equal to
    // output channel count for simplicity of the example.  In
    // reality, you would have to perform a channel mapping by hand.
    set_buffer_order(begin(processed_samples), end(processed_samples),
                     buffer.channels().size(),
                     buffer_order::interleaved,
                     buffer.get_ordering());
```

```
        // Copy the processed data into the output buffer.
        copy(begin(processed_samples), end(processed_samples),
             begin(raw_buffer));

        // Get a subspan for the remaining buffer
        auto remaining_buffer =
          raw_buffer.subspan(processed_samples.size());

        // Fill any remainder with zeros
        fill(begin(remaining_buffer), end(remaining_buffer),
             device::sample_type{});
    }
  });

  device.start();

  while(true); // Spin forever
}
```

8 Reference Implementation

A working repository with an implementation of the draft API that currently functions on macOS is available at https://github.com/stdcpp-audio/libstdaudio. This implementation is still a work in progress and may have some rough edges, but it does follow the API design presented here. We plan to get Windows and Linux implementations done after the February 2019 Kona meeting.

9 Polls for the Committee

We plan to present this paper in its current state at the February 2019 Kona meeting. Here are some polls that we would like to take at the meeting to receive guidance from the committee on how to proceed with this proposal.

9.1 General Polls

- Do we want to pursue audio for inclusion in the C++ standard?
- For the API design, do we want to pursue the direction presented in this paper?
- For the ship vehicle, do we want to pursue an Audio TS?

9.2 Underlying Type of a Buffer

In this paper, we have ignored the very important issue of the type of the samples in a buffer. Most consumer systems (MacOS, Windows, Linux) will happily mix floating-point values, but those same systems can be configured to mix with signed integers of varying widths.

Similarly, embedded hardware may have a restriction of using 8-bit buffers, fixed-point buffers, or even some exotic type such as a floating-bar [Quilez].

For now, we have chosen to leave this issue unresolved by using the strawman `device::sample_type` name everywhere, and assuming that it is a floating-point value for now. There are various design options:

- Should the sample type be a template parameter on the device?
- Should the sample type be a template parameter on a lower level – the buffer_view or the strided_span?
- Should the sample type itself be a type-erased type that the user has to cast at runtime?

The difference between these polls is an important and subtle one. If the device gets the template parameter, then there could be a potential type conversion cost in order to coerce the type provided by the driver into the type requested by the template parameter. (Or a runtime failure if there is a mismatch.) If the buffer gets the template parameter, then the end-user code is responsible for detecting the type and switching the behavior, which makes user code more complex. This is even more true if the lowest-level type (the sample itself) is type-erased, but on the other hand in this case the API doesn't have to be templated at all.

9.3 Alternate Backends

Many operating systems provide different APIs for accessing the low-level audio driver. For example, on Windows you can use WASAPI or ASIO, and on Linux you have a choice between ALSA, OSS, and Pulseaudio. We want to be able to support multiple different backend implementations for a single system – even within a single process. (For example, have two `std::audio::devices` in a single process, one of which is communicating with a sound device via WASAPI and another which is communicating via ASIO.)

There are two main design options:

- Should this functionality be made available by making the backend type a template parameter on the device class?
- Should this functionality be made available through virtual functions on the device class?

Templating the device class on the backend type makes the API more complicated, and makes it hard for the user to choose between different backends at runtime. On the other hand, making the device a polymorphic class and using OOP patterns to solve this issue doesn't feel like a design that blends well with the rest of the C++ standard library.

10 Future steps

The draft audio specification defined by this paper and its follow-ups in future mailings is intended to go through a number of different phases:

- **Phase 1** - Low-level device access. This paper represents the beginnings of phase 1. The goal here is to create a foundational framework that the remaining phases can build upon.
- **Phase 2** - Audio file I/O and playback. Once the foundation is laid, we can start dealing with file I/O and playback. There has been no design work done in this space, but it will likely take the form of a suite of algorithms and data structures that can be plugged in to the device callback. Adding this is crucial, since playing sounds from a file, and recording microphone input into a file, are probably the two most common uses of an audio API.
- **Phase 3** - (optional) MIDI. In music software, an important subcategory of audio software, MIDI has been the dominant industry standard since the 80s as a protocol for communicating musical information. If there is interest from the committee, an API for MIDI could be added to the standard library and would be a great addition to the audio API discussed here.

In the more immediate future, there are several features that we have not yet added but that are potentially important to have in this API even for Phase 1:

10.1 Changing the Device Configuration

We have already added getter functions for essential parameters of an audio device, such as `get_sample_rate()` and `get_buffer_size()`. However, on many systems the user can actually configure those parameters, and many audio apps use this functionality. We will therefore need to add setters as well.

10.2 Detecting Device Configuration Changes

The configuration of an audio device can also change outside of an app while it is running. For example, the user can change audio settings such as the sample rate or the default format in the operating system’s preference pane. Furthermore, new devices can be plugged in, existing devices can be unplugged, and another device can be selected as the default output/input device. The app will need to get notification callbacks on all of these. We are not yet sure which kind of notification API would be the best fit and provide good enough consistency with the design of the standard library.

10.3 Combining Multiple Devices

All of this API is currently centered around operating a single audio device. However, systems exist with more than one audio endpoint[10], and some method of coordinating among multiple devices is important.

In particular, there are optimization opportunities on some platforms. For example, on Windows, WASAPI allows the user to create an Event which is triggered whenever the device is ready to receive or provide samples. Ordinarily, the user will call `WaitForSingleObject()` on that Event (which is how `device::wait()` would be implemented under the hood). However, on systems with multiple audio endpoints, it is more efficient to create all of their Events, and then make a single `WaitForMultipleObjects()` call on all of the Events. This call will trigger whenever the first Event is triggered, and can be called repeatedly to ensure that the calling thread is woken up with a minimum of overhead.

This functionality could either be built into `device`, or be a separate class.

10.4 Channel Convenience Naming

There are a number of conventions in the audio world regarding the names of channels, and the order that they appear in an interleaved buffer. For example, in an interleaved stereo buffer, the common standard is to have the left channel first, then the right channel. Similarly, there are standards for other channel configurations. We would like to provide some convenience enums for audio buffers so that you can say (for example) `bv[channel::left]` in order to access the left channel. In order to make this useful, we would also have to provide some way of accessing/setting the channel order configuration.

10.5 Clocks and Timestamps

Some use cases for an audio library require a notion of physical time and a clock, and a way to timestamp audio buffers, for example when using audio in combination with video data. On the other hand, some platforms might support audio, but might not support any way of relating audio processing to physical time. It is therefore still unclear to us whether (and how) to include such functionality.

10.6 Exclusive vs Shared Mode

On many systems (including Windows and MacOS among others), the audio device can be opened either in exclusive mode or in shared mode. In exclusive mode, the device belongs to the program that opened it – no other audio programs from the system will be able to communicate with the audio device while it is opened by another program. In shared mode,

---

[10] In fact, they are common. Most commodity PCs will have audio outputs to plug speakers into, as well as being able to drive audio on a monitor through an HDMI connection.

the operating system runs a mixer under the hood which allows multiple programs to communicate with an audio device simultaneously. The OS might also apply a master volume and other effects to the output before sending it to the device.

Each setting has different advantages and disadvantages. Exclusive mode provides lower latency because there is no operating system mixer, but it has stricter requirements on buffer format, can potentially fail if another program already has the device open, and does not allow other programs to play audio through the device at the same time. Contrariwise, opening a device in shared mode provides for a wider variety of supported buffer formats, generally doesn't fail, and plays well with other programs, but at the cost of higher latency. Typically, games and consumer programs will use shared mode, while pro audio applications will use exclusive mode.

This distinction is important, and several backends let the application choose in which mode it wants to open the device. We therefore must design a way to expose this functionality.

11 References

[CppChat] https://www.youtube.com/watch?v=OF7xbz8fWPg

[AudioLibs] https://en.wikipedia.org/wiki/Category:Audio_libraries

[Forum] https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/Hkdh02Ejx6s

[ADC] https://www.youtube.com/watch?v=1aGoJSvwZjg

[GAP] Game Audio Programming Principles and Practices, Edited by Guy Somberg. Published by CRC Press. ISBN 9781498746731

[Waveforms] https://pudding.cool/2018/02/waveforms/

[PCM] https://en.wikipedia.org/wiki/Pulse-code_modulation

[SoundCard] https://en.wikipedia.org/wiki/Sound_card

[Jain] https://arxiv.org/pdf/0805.1598.pdf

[Quilez] http://iquilezles.org/www/articles/floatingbar/floatingbar.htm

[Noise] https://en.wikipedia.org/wiki/White_noise