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 #: P1386R2
Date: 2019-06-17
Project: Programming Language C++
Audience: SG13, LEWG

"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 describe our proposed design for such an API, as well as examples how to use it. An implementation of the API is available online.

Finally, we mention some open questions that will need to be resolved, 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
1.3 Why What We Have Today is Insufficient 5
1.4 Why not Boost.Audio? 7
2 Earlier proposals **7**
3 Scope **8**
4 Target Audience **9**
5 Background **9**
5.1 What is Audio? 9
5.2 Representing Audio Data in C++ 10
5.2.1 Samples 10
5.2.2 Frames and Channels 10
5.2.3 Buffers 11
5.3 Audio Devices 12
5.4 Audio I/O 12
5.5 Audio Driver Types 13
6 API Design **13**
6.1 Design Principles 13
6.2 Design Overview 13
6.3 Class audio_buffer 14
6.4 Callback concepts 16
6.5 Device Selection API 16
6.6 Device Configuration Change Callbacks 17
6.7 Class audio_device 18
6.8 Class audio_device_io 22
7 Example Usage **23**
7.1 White Noise 23
7.2 Process on the Main Thread 23
7.3 Sine Wave 24
8 Reference Implementation **24**
9 Next steps **24**
9.1 Channel Names and Channel Layouts 25
9.2 Device settings negotiation API 25
9.3 Exclusive vs Shared Mode 25
9.4 Combining Multiple Devices 26

Acknowledgements **26**

References **26**

Document Revision History

R2, 2019-06-17:

- Updated discussion of the proposal's scope and roadmap
- Modified `audio_buffer` to allow for a non-contiguous underlying memory layout
- Removed buffer order and audio driver type template parameters
- Added device list change callback and device start/stop callback
- Added device configuration change callbacks
- Added timestamps for recording time and presentation time
- Renamed `audio_device_buffers` to `audio_device_io`
- Added `audio_device::has_unprocessed_io()` for single-threaded processing
- Removed `audio_device::get_supported_*` functions

R1, 2019-03-11:

- Improved discussion of motivation
- Addressed comments from LEWGI and SG13 in Kona
- Made `mdspan` underlying type of buffer, replacing `buffer_view` and `strided_span`
- Added notion of different audio driver (hardware abstraction layer) types.
- Made audio driver type, sample type, and buffer order template parameters
- Removed `buffer_list` in favour of `audio_device_buffers`
- Changed `device_list` to return an `optional<device>`
- Removed nested namespace `audio` in favour of prefixing names with `audio_`
- Editorial changes

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 proprietary cross-platform middleware. 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. Further, it is the task of the standard library to provide basic facilities that can otherwise not be portably written in standard C++. Adding low-level audio is a logical consequence of this direction, affirmed by the efforts to standardize filesystem, networking, etc.

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 the hardware that is common on existing systems, 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.

1.3 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 and/or proprietary.
- 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 (auto result = AudioObjectGetPropertyDataSize(
    kAudioObjectSystemObject, &pa, 0, nullptr, &dataSize)
    || dataSize != sizeof(AudioDeviceID); result != noErr)
  return result;

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

AudioDeviceIOProcID ioProcID;
if (auto result = AudioDeviceCreateIOProcID(
    deviceID, ioProc, nullptr, &ioProcID); result != noErr)
  return result;

if (auto result = AudioDeviceStart(deviceID, ioProc); result !=
noErr) {
  AudioDeviceDestroyIOProcID(deviceID, ioProcID);
  return result;
}

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

OSStatus ioProc(AudioObjectID deviceID,
               const AudioTimeStamp*,
               const AudioBufferList*,
               const AudioTimeStamp*,
```

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 = static_cast<float*>(pData);
  for (size_t i = 0; i < numFramesAvailable; ++i) {
```

``
              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 = static_cast<float*>(buffer.mData);
      for (size_t i = 0; i < buffer.mDataByteSize; ++i) {
        pDataFloat[i] = get_random_sample_value();
      }
    }
  }
  return noErr;
}
``
	``
	    pDataFloat[i] = get_random_sample_value();
	  }
	
	  pRenderClient->ReleaseBuffer(numFramesAvailable, flags);
	}
	
	Sleep((DWORD)(hnsActualDuration/REFTIMES_PER_MILLISEC/2));
	pAudioClient->Stop();
	``

In both 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.

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

JUCE

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

  ~MainComponent() override {
    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)
};
```

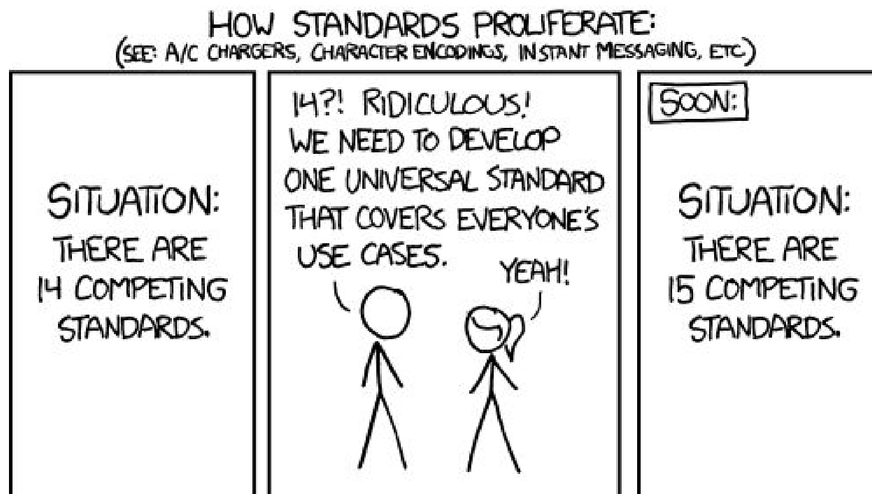
In this case, the abstraction is achieved by declaring a base class and requiring the client to override several member functions. While this does require less code, there is still redundancy in the form of four empty overridden functions, as well as a macro hiding a chunk of housekeeping. And, once again, our sample-generating (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 `auto` and `override` keywords 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 C++ interface to audio programming.

1.4 Why not Boost.Audio?

Obligatory reference to xkcd:



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

An oft-asked question is why we don't propose this library for Boost, instead of proposing it for the C++ standard library.

The short answer is that we are indeed planning to propose our reference implementation for Boost. It will be a great opportunity to get it into people's hands sooner while we go through the process of standardization. Additionally, libraries like Boost.Container show that there is room in Boost for filling in implementations for compilers that do not provide a complete standard library.

But submitting a library like this to Boost is orthogonal to submitting this proposal to ISO C++. C++ is still missing out-of-the-box access to a piece of fundamental and very common hardware.

2 Earlier proposals

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 devices 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.

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[2].

The scope of the paper is the set of common audio functionality that is not portably implementable in C++ today:

- Discovering what audio input/output devices are available on the system.
- Receiving callbacks when this configuration changes.
- Querying and configuring essential parameters of an audio device.
- A data structure for audio buffers that works across different systems.
- Performing audio I/O by reading/writing these buffers from/to an audio device.

This is the set that provides a minimal portable audio API layer opening the door to higher-level audio functionality. This paper does not seek to implement any such higher-level functionality at this point, such as audio file parsing/decompression/ playback, buffered streaming from disk or network, sound synthesis/effect algorithms, or a DSP graph[3]. It is worth pointing out that all of these can be implemented portably on top of what is proposed in this paper.

It is important to recognize that there exist forms of audio I/O that are not well represented by exchanging audio buffers with an audio device, the model used in this proposal. For example, spatial audio is an area of growing importance in the audio industry. There are systems such as Dolby Atmos, where the program renders audio to an encoded spatial format (instead of an output device), and then the OS renders that to the underlying audio hardware. As another example, mobile phones typically have a notion of *audio focus* and policies around how the audio focus comes and goes. An application producing audio output (such as a media player or a game) can be preempted by a higher-priority application (such as an incoming call). The audio focus might be returned to the app later when the call is over. Such platforms tend to have APIs based on audio sessions and policies instead of audio devices.

These use cases are all important, but also highly platform-specific. We do not think it is feasible to integrate them into a single standard C++ API that is supposed to behave roughly

[2] 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.
[3] DSP graphs are the most common way to represent complex audio processing chains.

the same across all platforms that support C++. These platform-specific use cases are therefore out of scope for this proposal.[4]

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 presented here are the same 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].

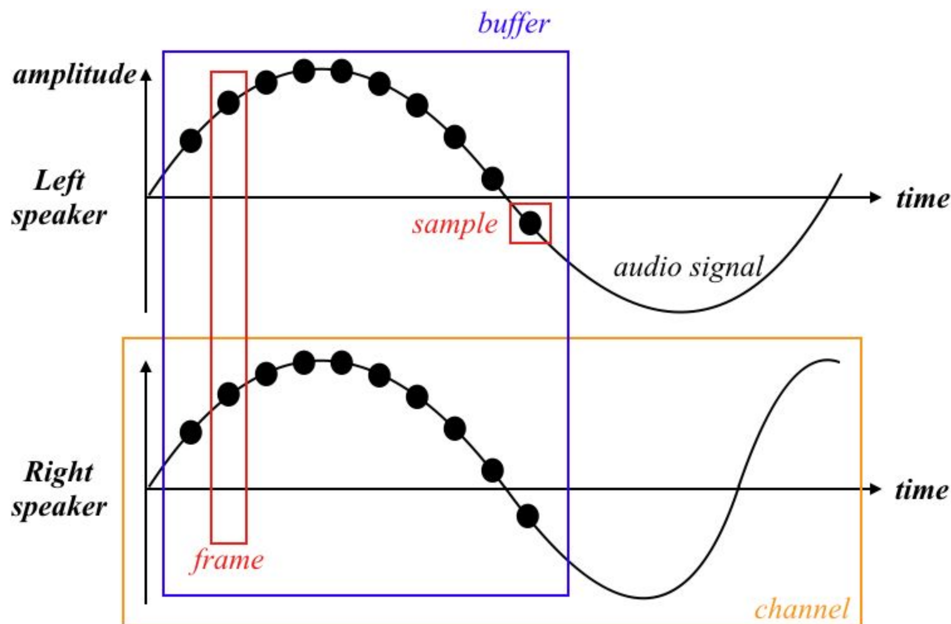
From chapter 1 of [GAP]:
"Any one-dimensional physical property can be used to represent air pressure at an instant in time. If that property can change over time, it can represent audio. Common examples are voltage (the most common analog), current (dynamic microphones), optical transitivity (film), 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."

---

[4] One thing to note is that many (probably nearly all) of these technologies still use the same sorts of techniques for populating a buffer with samples. The details occasionally differ slightly - maybe they provide multiple 3D-positioned streams (as in technologies like Dolby Atmos and Windows Sonic), or the underlying format is a compressed or otherwise encoded audio stream instead of PCM - but the principles underlying them are all very similar. As a result, if there is an interest in introducing a library that supports these paradigms, the patterns and techniques of this proposal will be invaluable.

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 block of consecutive frames and the most important data structure in audio (note how the word "buffer" has a different and more specific meaning in audio). 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 possible 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.

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

The audio buffer structure is equivalent to a 2D array of samples. A buffer is typically either contiguous in memory, or follows a pointer-to-pointer structure, although other memory layouts are possible. Typically, interleaved buffers are contiguous, which is optimal for operations like encoding/decoding an audio stream, while deinterleaved buffers are pointer-to-pointer, with each first-level-pointer pointing to a single channel, which is optimal for operations like applying an FFT to every channel.

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 I/O

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

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

---

[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.

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.

However, this paper does not propose to codify the restrictions of (near-)real-time computing in the C++ standard, or to explicitly define what kind of code can or cannot be called on the audio thread. In C++, such restrictions exist only implicitly.

5.5 Audio Driver Types

Many operating systems provide more than one API (hardware abstraction layer) for communicating with 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.

Different audio driver types offer different trade-offs. For example, ASIO offers lower latency than WASAPI, which is an important consideration for pro audio applications, but not all audio devices might come with an ASIO driver.

In a previous revision of this paper, the audio device and the device discovery API were all templated on the audio driver type. However we found that this complicates the API to an unacceptable degree. The present revision therefore no longer contains any facilities to use different driver types simultaneously in the same program.

6 API 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

The present revision (R2) covers a high-level design of the various classes and structures and their relationships. Future versions will include proposed wording and completely fleshed-out APIs.

Broadly, the proposed design provides abstractions for each of the concepts defined above.

- **`audio_buffer`** - A range of multi-channel audio data.
- **`audio_device_list`** - A list of currently available audio input or output devices.
- **`audio_device`** - A handle to communicate with an audio device through a hardware abstraction layer, and exchange audio buffers with it purposes of audio input and output (with a callback or by polling).
- **`audio_io`** - Represents the data exchanged by an audio device to the audio callback: the input and output buffers and associated timestamps.
- **`AudioDeviceListCallback`**, **`AudioDeviceCallback`**, and **`AudioIOCallback`** - Concepts for the three different types of callbacks used in the classes above.

The remainder of this section will go into greater detail about the specific classes, their usage, and their APIs. All classes and free functions are declared in the `std::experimental` namespace unless stated otherwise. We opted against a nested `audio` namespace for the reasons outlined in [P0816R0].

Note that the API description below is intentionally incomplete so that the signal does not get overwhelmed by the noise. For example, we have left out constructors, destructors, and thorough application of `const` and `noexcept` to name but a few items. These functions, as well as more formal wording to specify them, will come in future revisions of this document. Furthermore, all names in this proposal are placeholder names subject to future bikeshedding.

6.3 Class `audio_buffer`

The `audio_buffer` class provides access to audio data. It is essentially a view over a 2D array of samples. One dimension represents the channels, and the other represents the frames; the size is dynamic in both dimensions. The memory layout (contiguous, pointer-to-pointer, or otherwise) and the buffer ordering (interleaved/frames-first or deinterleaved/channels-first) is not specified, and `audio_buffer` can efficiently support any layout and ordering chosen by the implementation. However, member functions are provided to query whether either (or both) dimensions are laid out contiguously in memory, so that the user can write code optimized for a specific layout and ordering. This is inspired by the design of mdspan [P0009R9].

`audio_buffer` does not own its data, which is typically managed by the implementation. This is to avoid any unnecessary runtime overhead. `audio_buffer` objects are not meant to be instantiated by the user.

```
template <typename SampleType>
struct audio_buffer
{
  SampleType& operator()(size_t channel, size_t frame);
  const SampleType& operator()(size_t channel, size_t frame) const;
  SampleType* data() const noexcept;
```

```
  bool is_contiguous() const noexcept;
  bool channels_are_contiguous() const noexcept;
  bool frames_are_contiguous() const noexcept;

  size_t size_channels() const noexcept;
  size_t size_frames() const noexcept;
  size_t size_samples() const noexcept;
};
```

`SampleType& audio_buffer::operator() (size_t channel, size_t frames);`
Returns: A reference to the sample corresponding to a given frame and a given channel.

```
const SampleType& audio_buffer::operator() (size_t channel, size_t frame)
    const;
```
Returns: A const reference to the sample corresponding to a given frame and a given channel.

`SampleType* audio_buffer::data() const noexcept;`
Returns: A pointer to the beginning of the underlying contiguous array, or `nullptr` if `is_contiguous() == false`.

`bool is_contiguous() const noexcept;`
Returns: `true` if the underlying data is contiguous in memory.

`bool channels_are_contiguous() const noexcept;`
Returns: `true` if the buffer is deinterleaved, i.e. each channel is a contiguous array of samples in memory. In this case, a pointer to the i-th channel can be obtained from the expression `&buffer[0, i];`

`bool frames_are_contiguous() const noexcept;`
Returns: `true` if the buffer is interleaved, i.e. each frame is a contiguous array of samples in memory.

`size_t audio_buffer::size_channels() const noexcept`
Returns: The number of channels in the audio buffer.

`size_t audio_buffer::size_frames() const noexcept`
Returns: The number of frames in the audio buffer.

`size_t audio_buffer::size_samples() const noexcept`
Returns: The total number of samples in the audio buffer, equal to the number of channels multiplied by the number of frames.

`audio_buffer` is templated on the sample type, a numeric type representing a single audio sample. On some platforms, the native audio API might use sample types that cannot be easily represented as a numeric type in standard C++ (for example, 24-bit "packed" integers,

and integers with opposite endianness). On such platforms, the implementation can either choose to use a standard C++ type such as `int32_t` and perform type conversions under the hood, or use a custom "proxy" type for `SampleType` that gives access to the individual samples. All this is entirely implementation-defined; the standard API we provide does not impose any restrictions apart from `SampleType` being numeric. However, the most common sample type for audio APIs on the major operating systems is 32-bit `float` (followed closely by 16-bit `int`), so we expect that most programs will just use that.

6.4 Callback concepts

The proposed API requires three different types of callbacks, which are represented as concepts. This way, the functions that register a callback do not require any particular type of callable and will work the same with lambdas, function pointers, functors, etc. while still explicitly stating the required parameter types that each callback takes.

```
template <typename T>
concept AudioDeviceListCallback
  = std::is_nothrow_invocable_v<T>;
```
The callback used to react to changes in the device list, such as when a device is unplugged, a new one plugged in, or the default input or output device changes. Takes no arguments.

```
template <typename T>
concept AudioDeviceCallback
  = std::is_nothrow_invocable_v<T, audio_device&>;
```
The callbacks called by the device when it starts and stops. This type of callback can be used to do setup and teardown work before/after doing audio I/O.

```
template <typename T>
concept AudioIOCallback
  = std::is_nothrow_invocable_v<T, audio_device&, audio_device_io&>;
```
The audio I/O callback. Used to exchange audio data with an audio device. Takes a reference to an `audio_device`, and a reference to an `audio_device_io` structure holding the most recent input and/or output buffer.

For all these callbacks, it is implementation-defined on which thread they will be called. The nature of audio I/O also makes it unnecessary to add synchronisation facilities or explicit error handling to these callbacks or use executors, resulting in a very simple API.

6.5 Device Selection API

The first entry point to doing audio I/O with the proposed API is to figure out which device you wish to communicate with, using the device selection API.

```
optional<audio_device> get_default_audio_input_device();
```

Returns: a device object referring to the system-default audio input device, or no value if there is no default input device.

```
optional<audio_device> get_default_audio_output_device()
```
Returns: a device object referring to the system-default audio output device, or no value if there is no default output device.

```
audio_device_list get_audio_input_device_list();
```
Returns: a device list object to iterate over the input devices currently available on the system.

```
audio_device_list get_audio_output_device_list()
```
Returns: a device list object to iterate over the output devices currently available on the system.

```
class audio_device_list {
public:
  iterator begin();
  iterator end();
};
```

Platforms with no audio I/O capabilities can be represented in a straightforward way. On such platforms, `get_default_audio_input_device()` and `get_default_audio_output_device()` will always return optionals that hold no value. `get_audio_input_device_list()` and `get_audio_output_device_list()` will always return empty lists.

6.6 Device Configuration Change Callbacks

Whenever the device configuration changes, an application might need to react to that. In particular, the following three events are important:

- When the list of available devices changes (a device is unplugged or another device is plugged in)
- When the default input or output device changes (the current default device is unplugged, or the user chooses another default in the OS settings)

We offer an enum to tag each event, and a function to register a callback function for each event:

```
enum class audio_device_list_event {
  device_list_changed,
  default_input_device_changed,
  default_output_device_changed,
};
```

```
void set_audio_device_list_callback(audio_device_list_event event,
                                    AudioDeviceListCallback&& auto cb);
```

Effects: registers the given callback to be called whenever event occurs. It is unspecified on which thread the callback will be called by the system.

6.7 Class `audio_device`

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

```
class audio_device
{
public:
  string_view name() const;

  using device_id_t = /* implementation-defined */;
  device_id_t device_id() const noexcept;

  bool is_input() const noexcept;
  bool is_output() const noexcept;
  int get_num_input_channels() const noexcept;
  int get_num_output_channels() const noexcept;

  using sample_rate_t = /* implementation-defined */;
  sample_rate_t get_sample_rate() const noexcept;
  bool set_sample_rate(sample_rate_t);

  using buffer_size_t = /* implementation-defined */;
  buffer_size_t get_buffer_size_frames() const noexcept;
  bool set_buffer_size_frames(buffer_size_t);

  template <typename SampleType>
  constexpr bool supports_sample_type() const noexcept;

  constexpr bool can_connect() const noexcept;
  constexpr bool can_process() const noexcept;

  void connect(AudioIOCallback auto&& ioCallback);

  bool start(AudioDeviceCallback auto&& startCallback = no_op,
             AudioDeviceCallback auto&& stopCallback = no_op);
  bool stop();
  bool is_running() const noexcept;
```

```
  void wait() const;
  void process(AudioIOCallback auto& ioCallback&);
  bool has_unprocessed_io() const noexcept;
};
```

`string_view audio_device::name();`
Returns: The human-readable name of this audio device. There is no guarantee that this name is unique among the currently connected audio devices.

`audio_device::device_id_t;`
An implementation-defined type used for unique device IDs.

`device_id_t audio_device::device_id() const noexcept;`
Returns: a unique device ID that can be used to distinguish this device from all other currently connected audio devices of the same audio driver type.

`bool audio_device::is_input() const noexcept;`
Returns: `true` if the device is capable of audio input, `false` otherwise.

`bool audio_device::is_output() const noexcept;`
Returns: `true` if the device is capable of audio output, `false` otherwise.

`int audio_device::get_num_input_channels() const noexcept;`
Returns: The number of audio input channels that this device offers. Zero if the device is not capable of audio input.

`int audio_device::get_num_output_channels() const noexcept;`
Returns: The number of audio input channels that this device offers. Zero if the device is not capable of audio input.

`audio_device::sample_rate_t;`
An implementation-defined numeric type used to describe the device's sample rate.
Note: This can be either floating-point or integral, depending on the platform. Major desktop and mobile platforms tend to use either `double` or `unsigned int`.

`sample_rate_t audio_device::get_sample_rate() const noexcept`
Returns: The sample rate that the device currently provides to the program.
Note: This does not mean that the hardware device actually runs at that sample rate; the hardware abstraction layer is allowed to perform a sample rate conversion internally.

`bool  audio_device::set_sample_rate(sample_rate_t);`
Effects: Requests the device to use the passed-in sample rate for audio I/O.
Returns: A bool indicating whether the request was successful.
Note: This does not mean that the hardware device actually runs at that sample rate, and it is unspecified whether this will cause other clients of the same device to also observe a

sample rate change. This is to allow the hardware abstraction layer to perform a sample rate conversion internally, and supply audio data with a different sample rate to each client.

`audio_device::buffer_size_t;`
An implementation-defined integral type used to describe the device's audio buffer size.

`buffer_size_t audio_device::get_buffer_size_frames() const noexcept;`
Returns: The size of the buffer (in frames) that the device currently provides to the callback.

`bool audio_device::set_buffer_size_frames(buffer_size_t);`
Effects: Requests the device to use the provided buffer size (in frames) for the buffers supplied to the callback.
Returns: A bool indicating whether the request was successful.

`bool audio_device::is_running() const noexcept;`
Returns: whether the device is currently requesting (output devices) or generating (input devices) audio data.

```
template <typename SampleType>
constexpr bool audio_device::supports_sample_type() const noexcept;
```
Returns: `true` if this device can drive an I/O callback (via `connect` or `process`) with the specified `SampleType`.

`constexpr bool audio_device::can_connect() const noexcept;`
Returns: `true` if this device can drive a callback (via `connect`) on a separate thread created by the device.

`constexpr bool audio_device::can_process() const noexcept;`
Returns: `true` if this device can drive a callback via `wait`/`process`.

`void audio_device::connect(AudioIOCallback auto&& ioCallback)`
Mandates: `can_connect() == true.`
Effects: 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()` and before `stop()`, 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);
  }
}
```

```
bool audio_device::start(AudioDeviceCallback auto&& startCallback = no_op,
                         AudioDeviceCallback auto&& stopCallback = no_op);
```

Effects: Requests the device to start requesting (output devices) or generating (input devices) audio data. If `connect()` has been called on this device, this requests the audio device to start calling the callback on a separate audio processing thread. If the device has already been started, this call has no effect.
Arguments: Callbacks to be called by the implementation on an unspecified thread before the device starts calling I/O callbacks, and after calling the I/O callbacks has stopped, respectively.
Returns: A bool indicating whether the request was successful.
Note: The audio processing thread may be created before or after the call to `start()`). It may be created by the library or by the operating system. "Starting" the device does not mean that the underlying physical hardware device is initialized at this point, only that it starts exchanging data with this client – other clients may be already communicating with the same device.
Note: Some systems have policies in place for accessing audio devices. For example, an app might have to get permission from the user first before accessing the microphone. We expect that an implementation of this API would handle those things as required by the system. For example, the call to `start()` might bring up a user dialogue requesting the user to grant the required permission, and return `false` if that permission is refused.

```
bool audio_device::stop();
```

Effects: Requests the device to no longer request (output devices) or generate (input devices) audio data. If the device has not been started yet, or has been stopped already, this call has no effect.
Returns: A bool indicating whether the request was successful.
Note: "Stopping" the device does not mean that the underlying physical hardware device should shut down, only that this client is not interested in communicating with it anymore. Other clients may keep communicating with this device. Also, due to asynchronous nature of audio callbacks, the client might receive an audio callback even after a call to `stop()` has returned `true`.

```
void audio_device::wait() const
```

Mandates: `can_process() == true`
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, then this function returns immediately.

```
void audio_device::process(AudioIOCallback auto& cb);
```

Mandates: `can_process() == true.`
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.

`bool audio_device::has_unprocessed_io() const noexcept;`
Mandates: `can_process() == true.`
Returns: A bool indicating whether there is a new `audio_device_io` object that has not been processed yet. This is useful for doing work in between calls to `process()` on a single-threaded system where neither `wait()` nor `connect()` are meaningful.

6.8 Class `audio_device_io`

An object of type `audio_device_io` is a wrapper for the audio data exchanged with an audio device via an audio callback. It contains an optional input buffer and an optional output buffer and timestamps associated with the recording time and presentation time, respectively. This way we can accommodate input devices, output devices, and combined input/output devices with the same simple callback signature.

This wrapper is defined as follows:

```
using audio_clock_t = /* Implementation-defined */;

template <typename SampleType>
struct audio_device_io
{
  optional<audio_buffer<SampleType>> input_buffer;
  optional<chrono::time_point<audio_clock_t>> input_time;
  optional<audio_buffer<SampleType>> output_buffer;
  optional<chrono::time_point<audio_clock_t>> output_time;
};
```

`audio_clock_t;`
An implementation-defined audio clock type.

`optional<audio_buffer<SampleType> audio_device_io::input_buffer;`
An `audio_buffer` containing audio data that has been generated by the device, or no value if the device is not an input device.

`optional<chrono::time_point<audio_clock_t>> input_time;`
A timestamp representing the "recording time", i.e. the time when the first frame of the input buffer was physically recorded by the input hardware (such as a microphone).

`optional<audio_buffer<SampleType> audio_device_io::output_buffer;`
An `audio_buffer` that the device has requested to be filled with audio data, or no value if the device is not an output device.

`optional<chrono::time_point<audio_clock_t>> output_time;`
A timestamp representing the "presentation time", i.e. the implementation's best guess of the time when the first frame of the output buffer will be sent to the output hardware (such as a speaker).

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.

```
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() {
  auto device = get_default_audio_output_device();
  if (!device)
    return;

  device->connect([&](audio_device&, audio_device_io<float>& io)
  noexcept {
    if (!io.output_buffer.has_value())
      return;

    auto &out = *io.output_buffer;

    for (int frame = 0; frame < out.size_frames(); ++frame)
      for (int channel = 0; channel < out.size_channels(); ++channel)
        out(frame, channel) = white_noise(gen);
  });

    device->start();
    while(device->is_running());
}
```

7.2 Process on 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() {
  auto device = get_default_audio_output_device();
```

```
  if (!device)
    return;

  auto callback = [](audio_device& device, audio_device_buffers& buffers){
    /* ... */ };
  device->start();
  while(device->is_running()) {
    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[8], that's an A.)

```
int main() {
  auto device = get_default_audio_output_device();
  if (!device)
    return;

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

  device->connect([=](audio_device& device, audio_device_io<float>& io)
  mutable noexcept {
     auto& out = *io.output_buffer;

    for (int frame = 0; frame < out.size_frames(); ++frame) {
      float next_sample = std::sin(phase);
      phase = std::fmod(phase + delta, 2.0f * M_PI);

      for (int channel = 0; channel < out.size_channels(); ++channel)
        out(frame, channel) = 0.2f * next_sample;
  });

  device->start();
  while(device->is_runninng());
}
```

[8] 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.

8 Reference Implementation

A repository with a working 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.

This reference implementation also contains a test suite as well as example apps, demonstrating the example usages above as well as other features of our API like device enumeration and processing microphone input.

9 Next steps

This proposal is still missing some important features that we have not implemented yet. These will be added in future revisions.

9.1 Channel Names and Channel Layouts

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 use (for example) `channel::left` instead of a numeric index order to access the left channel.

More generally, the meaning of channels does not map to their index in the buffer in the same way on every platform. For example, Windows is using the channel order Left-Right-Center, whereas macOS is using Left-Center-Right. In order to portably send output to the Center channel, we need to provide a dynamic API for querying the current channel layout, as well as defining and requesting a set of channels we are interested in, which is a subset of the channels provided by the audio device. Some existing audio APIs provide this functionality by accepting a channel mask when opening an audio device. We do not yet have a proposal on how to standardise this functionality.

9.2 Device settings negotiation API

Device settings such as buffer size, sample rate, and the sample type highly depend on the particular device as well as each other. Some combinations of settings might be supported, while others might not. For this reason, many audio APIs such as WASAPI on Windows and ALSA on Linux offer a settings negotiation API, where you can query whether a particular combination of settings is supported, and more importantly, get a set of settings back from the driver that is different but as close as possible to the particular settings you requested. This revision of the paper does not offer such an API: you get to set the settings, start the audio device, and then register a "device started" callback in which you can query which

settings you actually got and react accordingly. This model is too limited for practical use compared to existing audio APIs. We therefore plan to add a proper settings negotiation API to the next revision of this paper.

In the last revision R1, we introduced the methods `get_supported_buffer_sizes_frames` and `get_supported_sample_rates` that return a `std::span` of the supported values for buffer size and sample rate, respectively. We have since found that this approach is not adequate (as the chosen setting for one might influence what settings are possible for the other), it does not include possible settings for the sample type, and it is not easily implementable for WASAPI.

9.3 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, 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.

9.4 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[9], 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 `audio_device::wait()` would be implemented under the hood). However, on systems with multiple audio endpoints, it is more

[9] 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.

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 `audio_device`, or be a separate class.

Acknowledgements

Many thanks to Gašper Ažman, David Hollman, and Christian Trott for their invaluable help with designing the `audio_buffer` class.

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

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

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

[P0009R9] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0009r9.html

[P0816R0] http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0816r0.pdf

[P1341R0] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1341r0.pdf