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 #: P1386R1
Date: 2019-03-11
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 4
- 1.4 Why not Boost.Audio? 6

2 Previous Work **7**

3 Scope **8**

4 Target Audience **8**

5 Background **9**
- 5.1 What is Audio? 9
- 5.2 Representing Audio Data in C++ 9
  - 5.2.1 Samples 10
  - 5.2.2 Frames and Channels 10
  - 5.2.3 Buffers 10
- 5.3 Audio Devices 11
- 5.4 Audio Software Interfaces 11
- 5.5 Audio Driver Types 12

6 API Design **12**
- 6.1 Design Principles 12
- 6.2 Design Overview 13
- 6.3 Class audio_buffer 13
- 6.4 Device Selection API 16
- 6.5 Class audio_device 17
- 6.6 Class audio_device_buffers 22
- 6.7 Algorithms 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 Roadmap **25**

10 Open questions **25**
- 10.1 Detecting Device Configuration Changes 25
- 10.2 Channel Names and Channel Layouts 26
- 10.3 Exclusive vs Shared Mode 26
- 10.4 Combining Multiple Devices 27

10.5 Clocks and Timestamps 27
10.6 Should We Use Executors and/or Coroutines? 27
10.7 Should Device Discovery Be a Centralized Facility? 28
10.8 What About non-Device-based I/O? 28

11 References 28

Document Revision History

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 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 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.
- 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*,
               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;
}
```

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) {
    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)
};
```

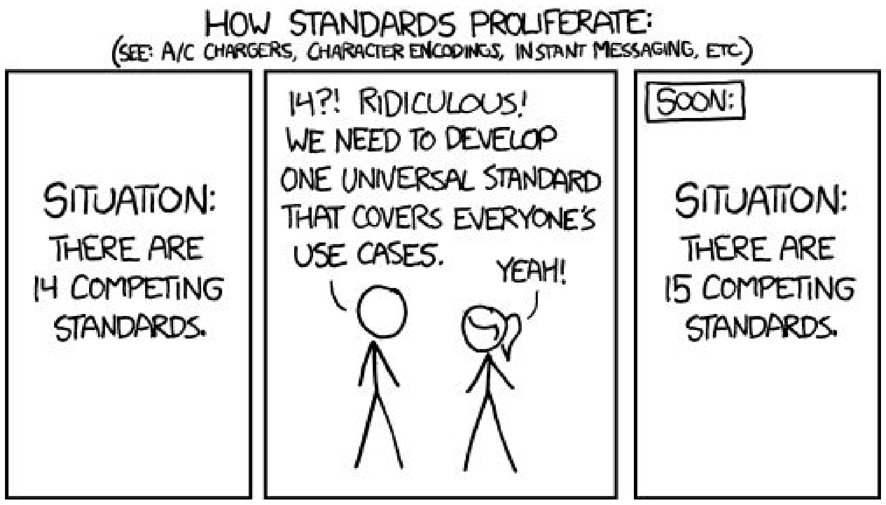
In this case, the abstraction is achieved by declaring a base class and requiring the client to override several member functions. While this does require less code, there is still redundancy in the form of four empty overridden functions, as well as a macro hiding a chunk of housekeeping. And, once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

1.4 Why not Boost.Audio?

Obligatory reference to xkcd:


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.
Source: https://xkcd.com/927/

Yeah, why not Boost.Audio? Why don't we submit this to Boost?

The short answer is that we probably should also submit this to Boost. It will be a great opportunity to get it into people's hands sooner while we go through the process of making an audio TS. 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 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 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.

Before publishing R0 of 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 the initial revision (R0) of this paper.

R0 was presented at the February 2019 WG21 meeting in Kona. We received encouragement to proceed with this proposal, as well as a large amount of technical feedback from both C++ library experts and audio domain experts. Based on this feedback, we made further adjustments to the API, which we present in this revision (R1).

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]. 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 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[3]. 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 (see also sections 9 and 10).

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.

[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.

5 Background

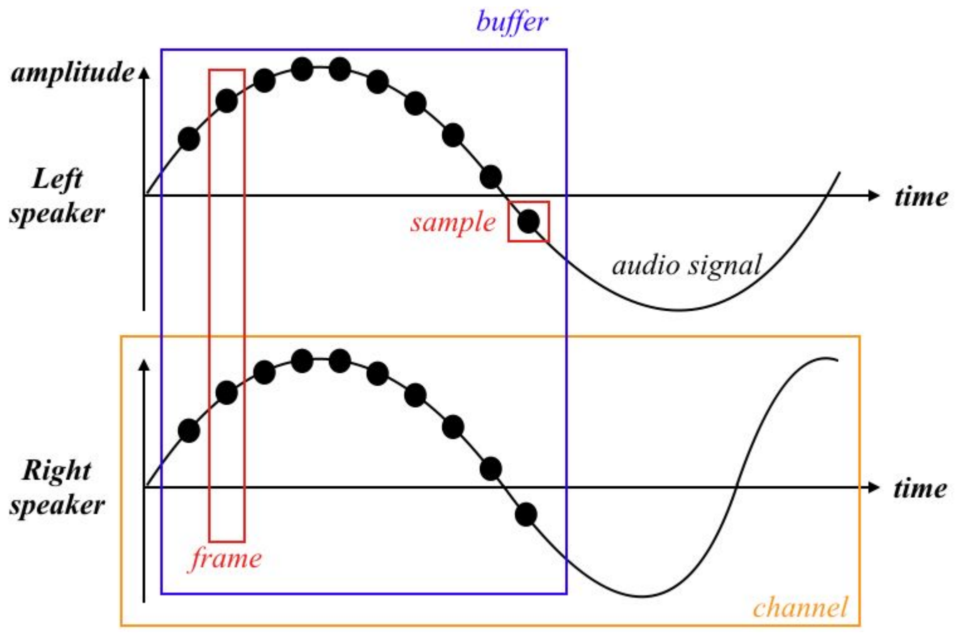
5.1 What is Audio?

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."

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[4]: 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[5]. 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[6] - 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: *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.

[4] 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.
[5] 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.
[6] 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.

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 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.

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.

In this paper, the term *audio driver type* refers to this kind of hardware abstraction layer.

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. Users who care about these tradeoffs must be given a choice which audio driver they want to use, and the ability to enumerate audio I/O devices separately for each such audio driver type – even within a single process.

For users who do not care about the audio driver type (the majority of users), there should be a default audio driver type. Furthermore, it is useful to consider "No Audio" to also be an audio driver type (one that does not offer any audio I/O devices).

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 (R1) 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 in Section 5 of this document:

- **`audio_buffer`** - A range of multi-channel audio data.
- **`audio_device`** - A handle to communicate with an audio device through a hardware abstraction layer, and exchange audio buffers with it through a device callback for purposes of audio input and output. Templated on the audio driver type.
- **`audio_device_list`** - A list of currently available audio input or output devices. Templated on the audio driver type.
- **`audio_device_buffers`** - Represents the data passed by an audio device to the audio callback. Provides an optional output buffer that the device is expecting to be filled (if it supports output), and an optional input buffer containing incoming audio data from the device (if it supports input).

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 this API 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 contiguous 2D array of samples, which is the prevalent way of representing such data. One dimension represents the channels, and the other represents the frames; the size is dynamic in both dimensions. The order in memory can be either interleaved (frames-first) or deinterleaved (channels-first). Furthermore, `audio_buffer` is a view – it does not own the data. As the underlying type of an `audio_buffer`, we choose an `mdspan` [P0009R9], which is exactly the type needed for this use case. To be more specific, the underlying type is

```
basic_mdspan<
  SampleType,
  extents<dynamic_extent, dynamic_extent>,
  BufferOrder>;
```

where `SampleType` is the numeric type representing a single sample value (integer, fixed-point, or floating-point), and `BufferOrder` is one of the following:

```
struct audio_buffer_order_interleaved {};
struct audio_buffer_order_deinterleaved {};
```

where "interleaved" corresponds to mdspan's `layout_left` and "deinterleaved" to `layout_right`, respectively.

The `audio_buffer` class itself provides a convenience wrapper around this specialization of `mdspan`:

```
template <typename SampleType = float,
          typename BufferOrder = audio_buffer_order_interleaved>
struct audio_buffer
{
  size_t size_channels() const noexcept;
  size_t size_frames() const noexcept;
  size_t size_samples() const noexcept;
  size_t size_bytes() const noexcept;
  SampleType* data() const noexcept;
  SampleType& operator()(size_t frame_index, size_t channel_index) const;
};
```

`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.

`size_t audio_buffer::size_bytes() const noexcept`
Returns: The total number of bytes in the audio buffer, usually equal to the number of samples times `sizeof(SampleType)`.

`SampleType* data() const noexcept;`
Returns: A pointer to the underlying raw data array.

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

This API has several properties that are worth pointing out.

On some platforms, the native audio API might represent audio buffers in a different way that doesn't result in a contiguous memory layout (for example, through a "pointer to pointers" where deinterleaved audio data is stored as an array of channels, and each channel is an array of samples). We cannot possibly provide a standard API that could account for all possible non-contiguous layout types. We therefore expect that on such platforms the implementation would convert their audio buffers to a format compatible with the contiguous mdspan layout (which, as far as we know, is the most common and practical way to represent audio data).

Further, 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.

For this reason, we also define `float` to be the default sample type. We also define interleaved to be the default order type. This is, to some degree, an arbitrary choice; we could just as well have picked deinterleaved (as far as we know, they are both equally common). However, by picking defaults for both, we can allow a user who doesn't care about either to not spell out any template arguments at all, which greatly simplifies the user code.

Finally, please note the lack of an iterator interface and `begin`/`end` functions in `audio_buffer`. The problem is that `audio_buffer` is a multidimensional array view (2D in this case). An attempt to define such an interface for a multidimensional view inevitably runs into the "view of views" problem, leading to an interface with iterators returning proxy objects instead of references, and the impossibility to define random-access iterators. This has already been investigated in depth by the authors of `mdspan` [P0009R9]. After experimenting with such "views of views" in R0 of this proposal, we decided that the best course of action is to be consistent with the `mdspan` API. Therefore, we now provide an `operator()` instead of an iterator interface for element access. The order of the indices in `operator()` corresponds to the natural order when looping through an interleaved buffer, which we chose as the default (see above).

Please note that when writing two nested loops over the channels and frames, respectively, it depends on the `BufferOrder` template argument which of the two possible ways will be the correct way around to loop over the data in contiguous memory order. However, the value of this argument is always known at compile time. This also means that such loops can be optimized by the compiler where possible.

6.4 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.

```
template <typename AudioDriverType = audio_default_driver_t>
optional<audio_device<AudioDriverType>> 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.

```
template <typename AudioDriverType = audio_default_driver_t>
optional<audio_device<AudioDriverType>> 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.

```
template <typename AudioDriverType = audio_default_driver_t>
audio_device_list<AudioDriverType> get_audio_input_device_list();
```
Returns: a device list object to iterate over the input devices currently available on the system.

```
template <typename AudioDriverType = audio_default_driver_t>
audio_device_list<AudioDriverType> get_audio_output_device_list()
```
Returns: a device list object to iterate over the output devices currently available on the system.

```
template <typename AudioDriverType = audio_default_driver_t>
class audio_device_list {
public:
  iterator begin();
  iterator end();
};
```

The device selection API is templated on the audio driver type, to offer the user access to all available audio driver types. The primary templates for all of the above functions are undefined. Implementations will offer specializations for the above functions for all audio driver types supported on the system. So, for example, on an Apple operating system there could be a `coreaudio_driver_t` specialization, and on Windows there could be both `wasapi_driver_t` and `asio_driver_t` specializations. This is entirely up to the implementation, and we do not propose to standardise any of these concrete specializations.

However, most users will only use some default driver type on the given platform, and won't be interested in the fact that multiple driver types are present. Therefore, the following type alias is used as the default template argument in the device selection API:

```
using audio_default_driver_t = /* implementation-defined */ ;
```

Further, we define the following type:

```
struct audio_null_driver_t {};
```

This driver type represents the absence of audio I/O. The functions above are required to provide specializations for this driver type. Those specializations always return an optional with no `audio_device` and an empty `audio_device_list`, respectively. Platforms with no audio I/O capabilities can then simply define `audio_default_driver_t` to be `audio_null_driver_t`, and not define any other audio driver types, to be conforming.

6.5 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.

```
template <typename AudioDriverType>
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;
  span<sample_rate_t> get_supported_sample_rates() const noexcept;
  bool set_sample_rate(sample_rate_t);

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

  template <typename SampleType, typename BufferOrder>
    constexpr bool supports_audio_buffer_type() const noexcept;
  constexpr bool can_connect() const noexcept;
  constexpr bool can_process() const noexcept;

  template <typename CallbackType>
    void connect(CallbackType);

  bool start();
  bool stop();
  bool is_running() const noexcept;

  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;

  template <typename CallbackType>
    void process(CallbackType&);
};
```

`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 integral 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.

```
span<sample_rate_t> audio_device::get_supported_sample_rates()
  const noexcept;
```
Returns: A range containing all values for the sample rate that this device supports.

`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.

```
span<buffer_size_t> audio_device::get_supported_buffer_sizes_frames()
  const noexcept;
```
Returns: A range containing all values for the buffer size (in frames) that this device supports.

`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, typename BufferOrder>
constexpr bool audio_device::supports_audio_buffer_type() const noexcept;
```
Returns: `true` if this device can drive a callback (via connect or wait/process) with the specified `SampleType` and `BufferOrder`. This might involve internal sample type conversions and buffer reorderings, causing additional runtime overhead.

```
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`.

```
template <typename CallbackType>
void audio_device::connect(CallbackType)
```
Mandates:

- `CallbackType` is a `CallableType` providing `operator()(audio_device&, audio_device_buffers<SampleType, BufferOrder>&);`
- `supports_audio_buffer_type<SampleType, BufferOrder>() == true;`
- `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();
```
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.
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.

```
template<class Rep, class Period>
void audio_device::wait_for(
  std::chrono::duration<Rep, Period> rel_time) const
```
Mandates: `can_process() == true`
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 audio_device::wait_until(
  std::chrono::time_point<Clock, Duration> abs_time) const
```
Mandates: `can_process() == true`
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.

```
template <typename CallbackType>
void audio_device::process(CallbackType&);
```
Mandates:
- `CallbackType` is a `CallableType` providing `operator()(audio_device&, audio_device_buffers<SampleType, BufferOrder>&);`
- `supports_audio_buffer_type<SampleType, BufferOrder>() == true;`
- `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.

6.6 Class `audio_device_buffers`

An object of type `audio_device_buffers` 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. 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:

```
template <typename SampleType, typename BufferOrder>
class audio_device_buffers
{
public:
  optional<audio_buffer<SampleType, BufferOrder>> input_buffer()
    const noexcept;
  optional<audio_buffer<SampleType, BufferOrder>> output_buffer()
    const noexcept;
};
```

```
optional<audio_buffer<SampleType, BufferOrder>>
audio_device_buffers::input_buffer() const noexcept;
```
Returns: 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<audio_buffer<SampleType, BufferOrder>>
audio_device_buffers::output_buffer() const noexcept;
```
Returns: 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.

6.7 Algorithms

Applications using this audio API might work with different sample types and different buffer orders, and so the need arises to explicitly convert between them. We offer a function that can convert from one buffer type to another, performing sample type conversion and/or (out-of-place) interleaving and deinterleaving as required:

```
template <typename SampleTypeFrom, typename BufferOrderFrom,
          typename SampleTypeTo, typename BufferOrderTo>
void convert_audio_buffer(
  const audio_buffer<SampleTypeFrom, BufferOrderTypeFrom>& from,
  audio_buffer<SampleTypeTo, BufferOrderTypeTo>& to);
```
Effects: Copies the audio data in the audio buffer `from` to the audio buffer `to`, converting samples to type `SampleTypeTo` (if different from type `SampleTypeFrom`), and reordering the buffer to `BufferOrderTo` (if different from type `BufferOrderFrom`).
Expects: Both buffers have the same size in frames.

In future revisions, we might add other algorithms if they are considered to be essential for a standard audio API.

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();
  device.connect([](audio_device& d, audio_device_buffers& buffers) {
    for (auto sample : buffers.output_buffer().data())
      sample = get_random_sample_value();
  });

  device.start();
  while(true); // Spin forever
}
```

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();
  auto callback = [](audio_device& device, audio_device_buffers& buffers){
    /* ... */ };
  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[7], that's an A.)

```
int main() {
  auto d = get_default_audio_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([=](audio_device&, audio_device_buffers& buffers) mutable {
    auto& out = buffers.output_buffer();
    for (int i_frame = 0; i_frame < out.size_frames(); ++i_frame) {
      auto next_sample = std::sin(phase);
      phase += delta;
      for (int i_ch = 0; i_ch < out.size_channels(); ++i_ch)
        out(i_frame, i_ch) = next_sample;
    }
  });

  device.start();
  while(true); // Spin forever
}
```

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. We plan to get Windows and Linux implementations done in time for the next WG21 meeting in July 2019 in Cologne. Implementations for mobile platforms will follow afterwards.

[7] 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.

This reference implementation also contains additional example apps, demonstrating features like device enumeration and processing microphone input that are not covered by the code examples above.

9 Roadmap

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. Similarly to audio I/O, MIDI I/O is not portable today, because it relies on platform-specific APIs to communicate with a MIDI interface. 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.

10 Open questions

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

10.1 Detecting Device Configuration Changes

The configuration of an audio device, as well as the set of available audio devices can 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. On some systems, for example mobile devices, such notifications will arrive quite frequently.

This turns out to be surprisingly complex because there are no existing standard library facilities that perform this sort of push notification. Low-level audio APIs such as WASAPI and CoreAudio typically report device changes on an unspecified thread, which may or may not be the audio callback thread. Existing higher-level audio APIs will typically note the change in an atomic value, and then report changes through a callback on an event loop or update() call. On the other hand, environments with no thread support at all are also important.

It is clear that we need to add some kind of notification API in order for a standard audio API to be useful. However, we are not yet sure which kind of notification API would be the most appropriate for all these aspects. Efforts to unify asynchronous APIs in C++ have started only very recently (see [P1341R0]).

10.2 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.

10.3 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.

10.4 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[8], 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 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.

10.5 Clocks and Timestamps

Some use cases for an audio library require a notion of physical time and a clock, for example syncing audio to a video stream or to events triggered by the program. Some platforms provide timestamped audio buffers for these use cases. Such a timestamp typically gives an indication of when a buffer will be sent to the output device, taking into account the current system latency.

On the other hand, some platforms might support audio, but no way at all to relate audio processing to physical time. It is therefore still unclear to us whether (and how) to include clock and time functionality in a standard audio API.

10.6 Should We Use Executors and/or Coroutines?

Low-level audio APIs typically use audio callbacks for I/O, and our proposed standard audio API does the same thing. There is clearly some overlap with executors, which aim to provide a basic building block for asynchronous execution in C++. Coroutines also seem useful: they allow to rewrite callback-based code such as our own sound-generating examples above in a much more concise and elegant way. However, at the time of writing, Coroutines have only just been merged into the C++20 working draft, and Executors are postponed until at least C++23. We have no experience yet with how any of these things would integrate into an audio API. For the time being, we prefer to follow the established practice of existing audio APIs, which are callback-based as presented here.

[8] 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.

10.7 Should Device Discovery Be a Centralized Facility?

If the C++ standard library will get some sort of hardware device discovery API in the future, we should make sure that audio is a part of that system. We also want to make sure that the audio device discovery is independent of other classes of devices. For example, you wouldn’t want an interleaved list of network and audio devices. However, at present we are not aware of any published proposals in this space.

10.8 What About non-Device-based I/O?

There are plenty of audio I/O use cases where the model of exchanging audio buffers with an audio device, as discussed here, does not apply very well.

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, some systems have a notion of *audio focus* and policies around how the audio focus comes and goes. For example, on a mobile phone, an application producing audio output (such as a media player or a game) can be preempted by a higher-priority application (for example, 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.

It is currently unclear to us whether and how we would want a standard C++ API to cover such use cases.

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

[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