Appendix B
Introduction to Windows Core Audio

There are quite a few options when writing audio software for the Microsoft Windows platform. Perhaps you've heard of APIs like WinMM (aka, MME), DirectSound, XAudio2, etc. For the uninitiated, the number of choices can feel a bit overwhelming and it's not always clear where to start [1].

In this book, the Windows implementations of all the examples are built on top of a set of APIs collectively referred to as Core Audio[2]. Core Audio is a low-level audio API that was introduced in Windows Vista and continues to serve as the foundation for other audio APIs on modern Windows platforms[3].

Core Audio is a thin layer of abstraction designed with low-latency and reliability in mind. For performance reasons, it's implemented entirely in native code. You'll find no managed components within Core Audio.

Other audio APIs, such as XAudio2 and DirectSound, have a lot of bells and whistles baked in. Those libraries implement and abstract away many of the concepts I try to teach you in this book. I don't want to discourage you from learning other audio APIs. In your travels, you'll undoubtedly find occasions where other audio APIs are more appropriate. But for the purposes of this book, Core Audio makes the most sense.

This chapter provides a gentle introduction to Core Audio. It's not comprehensive by any stretch. There's simply not enough room in a single chapter for that. But this chapter should give you enough foundational knowledge to begin writing your own Core Audio application, as well as a solid understanding of what the book's audio library is doing under the hood.

[1] A thorough discussion of the history of Windows audio APIs can be found on my blog - http://www.shanekirk.com/2015/10/a-brief-history-of-windows-audio-apis/

[2] Not to be confused with Apple's Core Audio, which is a completely different API that accomplishes much of the same things for iOS and OSX.

[3] At least that's true of Windows 10, which is the latest version as of this writing.

B.1 Getting Started

Before we roll up our sleeves and begin writing code, it's important that we get a few introductions out of the way. This section starts with a short discussion on prerequisites, coding conventions, and some assumptions I make throughout the rest of the chapter. We'll then look at the Windows audio architecture, introduce a few related terms and concepts, and conclude this section with a quick introduction to the Core Audio APIs.

Learning to Live With COM

Believe it or not, there's more to native Windows programming than Win32. Many interesting things can only be accomplished with COM. That's a pill many developers new to Windows programming have a hard time swallowing. Core Audio, like many other APIs and component libraries, is built around COM. If you've been avoiding learning COM or have had no exposure to it, you should probably bite the bullet and at least learn the basics. This is especially true if you foresee spending any significant time of your career developing native applications for Windows. To get the most out of this chapter, you'll need at least a basic understanding of COM.

COM is an acronym that stands for Component Object Model. Essentially, it's a mechanism that allows code-sharing between different libraries, different processes, and even different machines. I can't teach you the basics of COM. I'd like to, sure. But there's a lot to learn and I wouldn't do the material justice in this short space. If you're new to COM, I recommend picking up a book on the topic, reading it, and then revisiting this material when you're more comfortable with it. The canon of COM books is Don Box's "Essential COM". It's an old book, but it's also a classic. COM technology hasn't changed that much over the years. So you can rest assured knowing that everything you read in that book is still applicable to modern COM development.

A Note Regarding Coding Conventions

Many C/C++ oriented books that deal with COM oriented technologies litter their example code with explicit calls to cleanup functions (e.g., CoTaskMemFree(), PropVariantClear(), IUnknown::Release(), etc.). Much of that is to reinforce concepts. Some of it is because they're using C and have no other options. This book, however, targets C++ developers. As C++ developers, we should enjoy the benefits the language has to offer. One of these benefits is

RAII[4]. In this chapter, I make heavy use of RAII classes such as smart pointers and object wrappers. Their use improves the code readability and promotes exception-safety.

_com_ptr_t

On of the smart pointer types I use throughout this chapter is _com_ptr_t. For those unfamiliar with _com_ptr_t, it's a template class provided by the platform SDK and is used for managing the lifetime of a COM object.

In practice, it's rare to see the _com_ptr_t type appear explicitly in code. That's because the platform SDK provides a convenient macro, _COM_SMARTPTR_TYPEDEF, for declaring typedefs for com_ptr_t specializations. These typedefs are a tad more programmer-friendly and easier to read.

The _COM_SMARTPTR_TYPEDEF macro accepts two arguments – the name and UUID of the interface you're interested in. It declares a smart pointer type with the same name as a given interface name, but with a "Ptr" suffix. In most cases, the smart pointer can be used in place of a raw pointer. Listing B.1 shows an example of how this works using interfaces that we discuss later in the chapter.

Near the top of the example, _COM_SMARTPTR_TYPEDEF declares two smart pointer types – IMMDeviceEnumeratorPtr and IUnknownPtr.

Next, an instance of IMMDeviceEnumeratorPtr is created, which we call spDeviceEnumerator. By default, this object points to nothing. It's effectively equivalent to a null interface pointer. Don't worry about what IMMDeviceEnumerator is used for. We'll get into all the gory details of that later on in the chapter.

Because _com_ptr_t objects have pointer semantics, we can use them with CoCreateInstance() just as we would a naked COM pointer. In the example, CoCreateInstance() is used with spDeviceEnumerator to acquire a reference to an object of type IMMDeviceEnumerator. This looks just as it would if we were using a naked interface pointer.

[4] RAII, or Resource Acquisition Is Initialization, is a horribly named idiom for managing resource lifetime. It's a simple idea you're surely already familiar with – using scope and object lifetime to manage the lifetime of some other resource. As far as C++ best-practices go, RAII is at the top of the list.

Listing B.1 – _com_ptr_t example

```
#include <comdef.h>      // Contains the macro definition
#include <Mmdeviceapi.h> // Provides the MMDevice API IIDs

_COM_SMARTPTR_TYPEDEF(IMMDeviceEnumerator, __uuidof(IMMDeviceEnumerator));
_COM_SMARTPTR_TYPEDEF(IUnknown,            __uuidof(IUnknown));

// Creating a COM object.
IMMDeviceEnumeratorPtr spDeviceEnumerator;
if (FAILED(CoCreateInstance(__uuidof(MMDeviceEnumerator), NULL, CLSCTX_ALL,
    __uuidof(IMMDeviceEnumerator), (void **)&spDeviceEnumerator)))
{
    return;
}

// Obtaining a different interface pointer.
IUnknownPtr spUnknown;
if (FAILED(spDeviceEnumerator.QueryInterface(__uuidof(IUnknown), &spUnknown)))
{
    return;
}

// Assignment from one _com_ptr_t to another. Automatically calls
// AddRef()
IUnknownPtr spUnknown2 = spUnknown;

// When _com_ptr_t instances go out of scope, Release() will be called
// on the underlying COM objects.
```

Next, we demonstrate the use of _com_ptr_t's QueryInterface() method to obtain a reference to IUnknown, which is also stored in a smart pointer. _com_ptr_t's QueryInterface() method simply delegates to the underlying object's QueryInterface() method. We could have just as easily called the underlying object's QueryInterface() method directly and achieved the same result.

We then assign from one smart pointer to another of the same type. Using raw pointers would have required us to invoke AddRef() on the COM object to make sure lifetime is properly managed. However, because we're using smart pointers, AddRef() is invoked automatically on our behalf.

Note the lack of explicit calls to Release() at the end of Listing B.1. Those weren't left out on accident. It turns out, we don't need to explicitly invoke Release() on objects of type _com_ptr_t. When our smart pointers go out of scope, Release() will be invoked on the underlying object automatically as part of the _com_ptr_t destruction process.

A Brief Word on UWP and WinRT

Windows 8 was developed and released during an era of transition for modern computing. PCs were no longer the only kid on the block. Windows phones, tablets, and "Internet of Things" devices had long since found places in our homes, cars, and places of work.

Traditional native Windows applications were built special for each type of device the author wanted to support. For example, if an application developer intended to support both the desktop PC and Windows Mobile, that usually meant developing two different applications with similar appearances and functionality.

Windows 8 attempted to bridge the gap. It introduced us to a new application framework called WinRT[5] (aka, Windows Runtime). And with Windows 10 came WinRT's successor, the UWP (Universal Windows Platform).

Both WinRT and UWP are native APIs for building applications that can target multiple Windows platforms. Using one of these, a single application can be built such that it runs on Windows desktop PCs, Windows phones, Surface tablets, and Windows IoT devices. This would have been no small feat prior to Windows 8.

What do these have to do with Core Audio? Well, at the time of this writing, applications built using WinRT or UWP only have access to a subset of offerings from Core Audio. Things like the MMDevice and DeviceTopology APIs, are mostly off limits, plug-in support is limited, and MIDI support was only introduced as of Windows 10.

We'll undoubtedly see the landscape change as this technology evolves. But for now, I expect most professional audio applications will continue to be built as traditional Win32 desktop applications.

[5] Not to be confused with Windows RT, the ARM based version of Windows. Like Passport and .NET, RT is another branding overload from the marketing folks at Microsoft.

The Windows Audio Architecture

Let’s now turn our attention to the Windows audio architecture. Figure B.1 shows a simplified representation of how rendered audio data flows from most[6] applications to the speakers. For captured audio, the path taken by the audio data looks looks exactly the same, but flows in the reverse direction.

The top of the diagram shows applications built around different audio technologies. In addition to a Core Audio application, the diagram depicts applications built using DirectSound, MME, and XAudio2. Note that this isn't intended to be a complete list of Windows audio technologies. The other APIs are shown here merely for context.

From Figure B.1, you can see that applications using other audio APIs are actually using Core Audio behind the scenes. Since Core Audio was introduced in Windows Vista, all other officially supported audio APIs have been built on top of Core Audio. Why use one of those other APIs if Core Audio is just going to be used anyway? As mentioned in the chapter's introduction, Core Audio is a fairly low-level API. Depending on your application's needs, it may be easier to accomplish certain types of things using one of the higher-level APIs, such as XAudio2. They provide abstractions that are sometimes easier to work with and features that you'd have to implement yourself in a Core Audio application.

From the diagram, you can see that audio data passes through the Core Audio APIs by means of an audio data buffer, sometimes referred to as an audio endpoint buffer. The audio data buffer contains a collection of byte data that originates in your application and is ultimately processed by the audio engine and/or audio device.

As you can see from Figure B.1, the data in the audio buffer can take two possible paths. Which path it travels depends entirely on the choice of streaming mode, of which there are two – shared-mode streaming and exclusive-mode streaming.

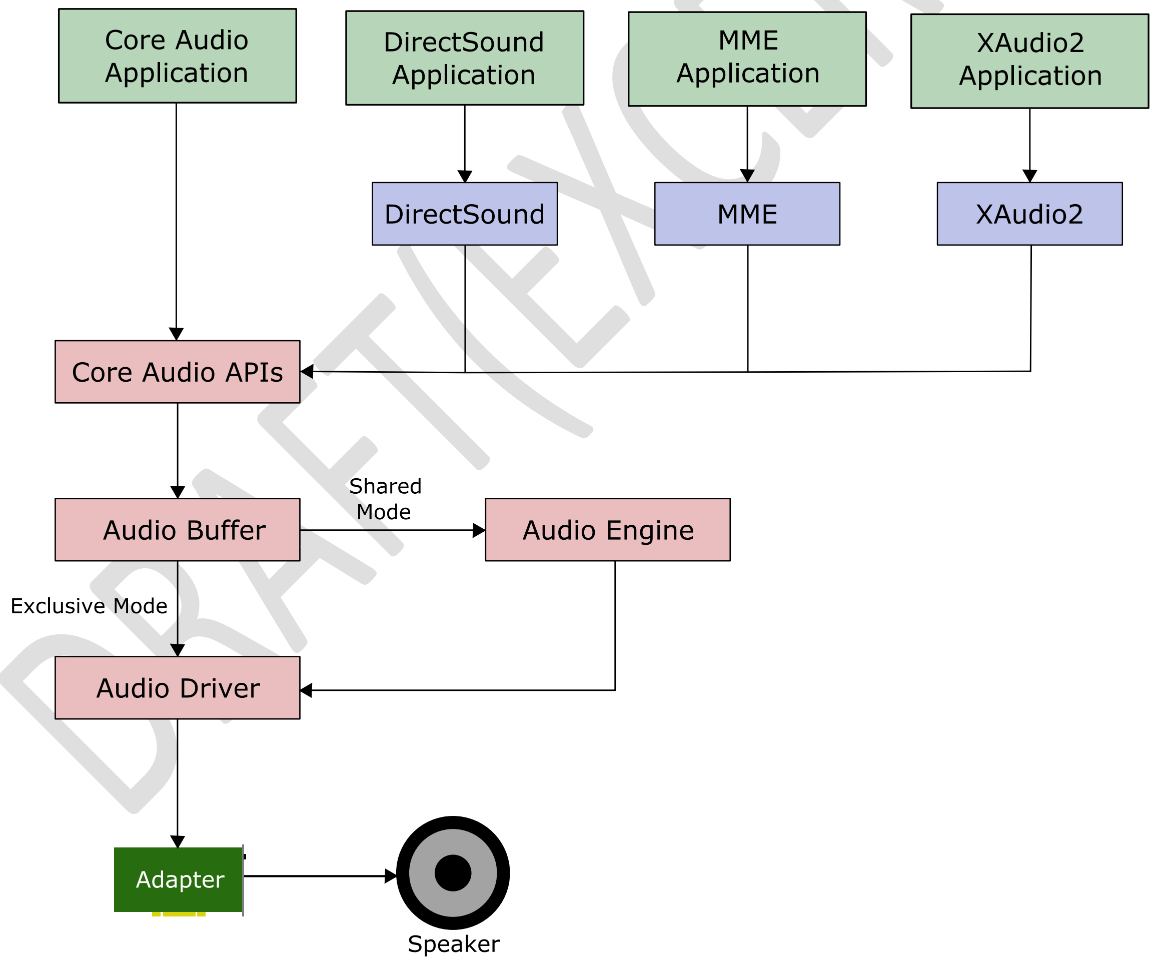
Shared mode streaming allows an application to play nice with other audio applications that may be running on the system. It’s not uncommon for multiple applications to render audio at the same time. When you play music or video in your web browser, for example, you're usually still able to hear Windows' system sounds and alarms as you click around on things. To make this work, Windows employs something called the audio engine.

[6] Using older techniques like WDM kernel streaming or third party technologies like ASIO, it's actually possible to circumvent the Windows audio system entirely and speak directly to the driver. We don't concern ourselves with those types of things in this book. Core Audio, and its support for exclusive-mode streaming, solve most of the problems the aforementioned techniques were invented to address.

There’s only one audio angine in Windows and it's shared by all applications that render or capture audio using shared-mode streams. That includes Windows itself. In the case of rendered audio, the audio engine takes the data from running applications, mixes it all together, and delivers a single audio stream to the audio driver. The audio driver then delivers the digital data to the audio adapter, which converts it to analog signals that get sent to your speakers or headphones. For captured audio, the process works entirely in reverse.

An advantage of using shared-mode audio is that you have some flexibility with audio formats. In many cases, the audio engine can automatically convert an audio format that you’d like to use to whatever format the audio engine is currently using. It's even possible to use formats that the audio adapter may not even natively support. We’ll discuss all of this in more detail later when we talk about rendering audio in Section B.5.

Figure B.1 - The Windows Audio Architecture simplified



The biggest disadvantage of shared-mode streaming is latency. Not only does the audio engine sometimes need to convert audio data, but it also has to mix data from multiple shared-mode applications. This takes time. Milliseconds, usually. In many cases, this latency isn't even perceivable. However, there are some cases where this latency can be problematic. The user experience in pro-audio applications and some types of games, for example, can be so severely impacted that it feels unusable.

Exclusive-mode streaming bypasses the Windows audio engine entirely. It effectively locks out all applications but one from being able to deliver audio data to the adapter, including the Windows audio engine. All audio data passing through the audio engine is effectively dropped when an application is using exclusive-mode streaming.

An obvious advantage of exclusive mode audio is that with the audio engine out of the picture, the latency[7] it would otherwise introduce is completely eliminated. This is a big win for pro-audio applications[8].

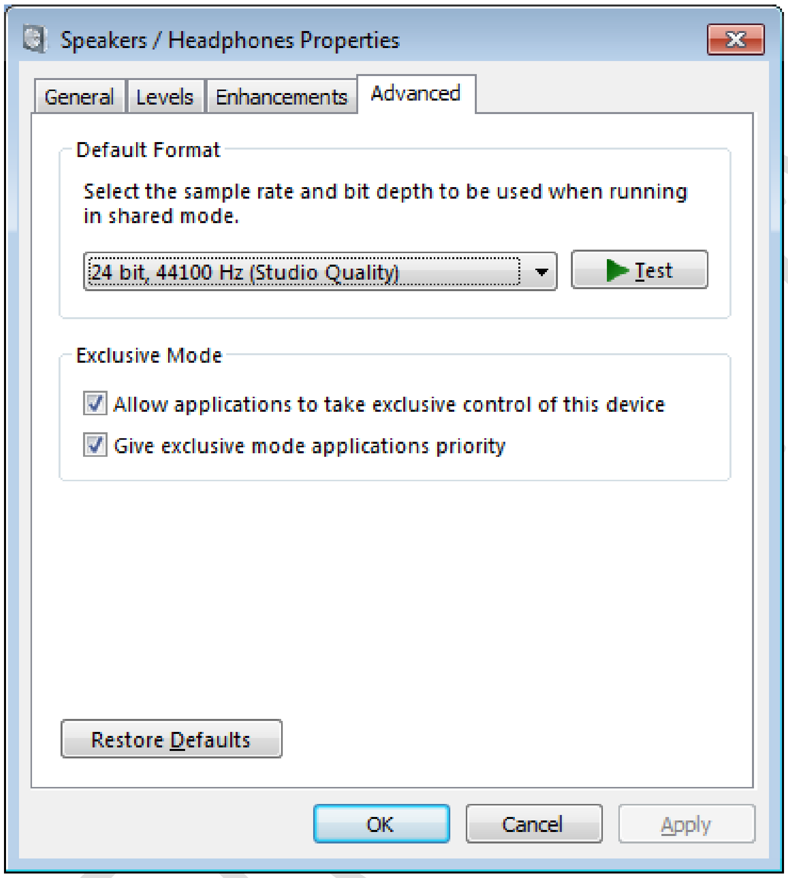
The biggest disadvantage for exclusive-mode streaming is that you don't have much flexibility with audio formats. You can only use a format that the audio adapter supports natively. If data format conversions will need to be performed, your application will need to do this manually.

It's worth pointing out that exclusive-mode streaming isn't actually guaranteed to be available to applications. It's user configurable. The user can completely disable exclusive mode audio for a given audio adapter. Figure B.2 shows an example of the speaker/headphone configuration dialog. Notice the option in the middle for completely disabling exclusive mode. The exact look and feel of this dialog may vary from system to system.

[7] Another type of latency that can be problematic on Windows is referred to as DPC Latency, or Deferred Procedure Call Latency. Due to the way Windows drivers work, one slow driver can impede another driver's ability to process data in a timely fashion. Windows 10 introduced "Audio Core Isolation" to combat this. At present, however, it's an opt-in feature for audio driver vendors.

[8] Before Core Audio, pro-audio applications would go to great lengths to bypass the Windows audio system. A technique popularized by Cakewalk was something called WDM Kernel Streaming (WDM-KS). Using WDM-KS, applications could stream data directly to the WDM audio drivers. It was kludgy, but effective. WDM-KS has since been made obsolete with the introduction of Core Audio's exclusive-mode streaming.

Figure B.2 - Screenshot of an example Speaker/Headphone Configuration dialog


Speakers / Headphones Properties

General | Levels | Enhancements | Advanced

Default Format

Select the sample rate and bit depth to be used when running in shared mode.

24 bit, 44100 Hz (Studio Quality)

Exclusive Mode

☑ Allow applications to take exclusive control of this device

☑ Give exclusive mode applications priority

Adapters, Drivers, Endpoints, Oh My!

In the previous discussion, we saw the paths audio data can take once it leaves your application, all roads ultimately leading to the audio adapter. And while this isn't technically inaccurate, it doesn't tell the whole story.

Audio adapters rarely have a single input and/or output connection. In fact, most modern consumer PCs ship with audio adapters that support at least three types of connections - headphones, speakers, and microphone. Professional grade audio adapters can have many times that. So, to say that a rendered audio buffer's final destination is the audio adapter says nothing about which output jack, port, or channel is actually being used.

When developing Core Audio applications, you don't usually think about audio output in terms of adapters or drivers. Everything is considered an *audio device*. For example, an audio adapter is an audio device. But so are the speakers, microphones, and headphones that plug into that adapter. A TASCAM 16-channel USB audio interface is an audio device. And so is each input and output channel on that interface.

But, wait a minute? If everything is a device, how do you distinguish between an adapter and its inputs and outputs? Input and output audio devices are referred to by another name – *audio endpoint devices*. These are the devices that serve as the initial source or final destination for audio data. In terms of physical hardware, everything is an audio device, but not everything is an audio endpoint device.

The whole notion of audio devices vs. audio endpoint devices might seem like a persnickety detail. Core Audio, however, is careful to distinguish between the two. And you'll find this distinction made throughout the Core Audio documentation on MSDN.

Audio Streams

We've already seen the phrase audio stream thrown about a few times in this chapter, but what exactly is an audio stream? How do you create one? And how do use it?

An audio stream is merely a connection between your application and an audio endpoint device. In the case of exclusive-mode audio streams, an application has a direct connection to an audio endpoint device. With shared-mode audio streams, audio data takes a detour through the Windows audio engine.

At the heart of the audio stream is a data buffer. One end of the stream is responsible for putting data into the data buffer, while the other end is responsible for reading data back out. This back and forth requires a bit of coordination between the two ends of the stream, and as you'll see later in the chapter, this is where things can sometimes get tricky.

Applications can create and use more than one audio stream. In fact, in some instances they have to. Audio data can only flow in one direction per stream. Applications that perform both rendering and recording need to create at least two streams – one stream for rendering audio and one stream for capturing audio.

Audio Sessions

Core Audio was designed around the concept of an audio session. The API used for rendering audio, WASAPI, even has "session" as part of its name – Windows Audio Session API. Ironically, however, audio sessions aren't typically something that application developers need to think much about. I'll explain why momentarily. But first, let's talk about what sessions are and why you might care about them.