PortAudio – an Open Source Cross Platform Audio API

Ross Bencina, Phil Burk

email: rossb@audiomulch.com, philburk@softsynth.com

website: http://www.portaudio.com/

Abstract

This paper describes a new open-source cross-platform 'C' library for audio input and output. It is designed to simplify the porting of audio applications between various platforms, and also to simplify the development of audio programs in general by hiding the complexities of device interfacing. The API was worked out through community discussions on the music-dsp mailing list. A number of people have contributed to the development of the API and are listed on the web-site. Implementations of PortAudio for Windows MME and DirectSound, the Macintosh Sound Manager, and Unix OSS have been developed and are freely available on the web. Support for other platforms is being planned. The paper describes the use of PortAudio and discusses the issues involved in its development including the design philosophy, latency, callbacks versus blocking read/write calls, and efficiency.

1. Introduction

Suppose you want to write a real-time audio application and you want it to run on more than just one kind of computer. That could mean that you have to learn how to use several different audio APIs like DirectSound™ on Microsoft Windows™, and the Macintosh™ Sound Manager, and OSS on Linux. And then you would have to write interfaces between your program and each of those host-specific APIs. Wouldn't it be nice to just write one simple audio interface for your program and to have it work on the majority of computers. PortAudio is designed to help you do just that.

PortAudio is also useful if you are only writing for one platform because PortAudio is often much simpler to use than the native audio APIs. Thus PortAudio could be useful for pedagogical purposes by allowing beginning students to quickly obtain real-time audio input and output.

PortAudio provides a platform neutral interface to real-time audio streaming services in the form of a 'C' language API. PortAudio has been implemented for a number of platforms by wrapping native audio services - these implementations are publicly available under a BSD style Open Source license. PortAudio was selected as the audio component of a larger initiative called PortMusic that combines audio, MIDI, and audio file I/O.

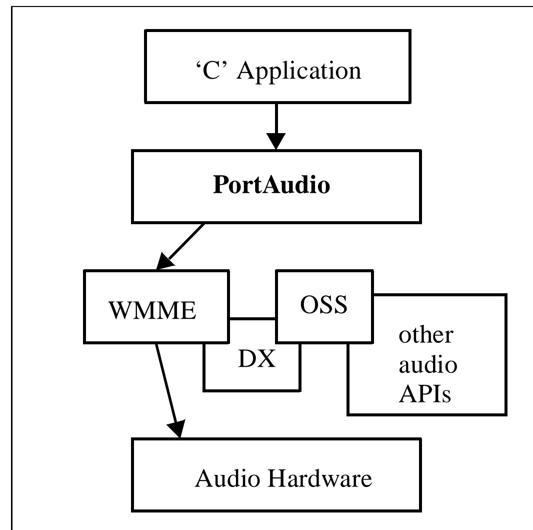

Figure 1, PortAudio generally uses existing audio APIs to access the audio hardware.

2. The PortAudio Architecture

The PortAudio architecture includes two main abstractions: Audio Devices and Audio Streams.

Audio devices represent audio input and/or output ports on the host platform. The PortAudio API provides functions for enumerating available devices and querying them for properties such as available sampling rates, number of supported channels and supported sample formats.

Audio streams manage active audio input and output through at most one input device and one output device - streams may be full duplex or half duplex. A PortAudio implementation manages buffers internally and requests audio processing from the client application via a callback that is associated with a stream when it is opened.

A variety of sample formats are supported by PortAudio including 16 and 32 bit integer and 32 bit floating point. Where necessary PortAudio manages conversion between the requested buffer formats and those available natively. If requested, PortAudio can clamp out of range samples and/or dither samples to a native format.

3. API Overview

This section presents an informal overview of the C language functions and structures that make up the PortAudio

API. Developers interested in making use of the API are advised to consult the portaudio.h header file, which contains full documentation for the API. There is also a tutorial on the PortAudio website.

A minimal PortAudio program that performs ring modulation and mixing of a stereo input signal can be found at the end of this paper. It does not do any error checking in order to save space. It, hopefully, demonstrates the simplicity of using PortAudio to obtain audio I/O on multiple platforms.

3.1. Initialisation and Device Enumeration

Before making use of the PortAudio library `Pa_Initialize()` must be called to initialise the library. When the library is no longer required `Pa_Terminate()` should be called.

`Pa_CountDevices()` returns the number of available audio devices. `Pa_GetDeviceInfo(id)` returns a pointer to a `PaDeviceInfo` structure which contains information about the requested device. It includes the device name, maximum number of input and output channels, available sample rates, and the supported data formats.

The `Pa_GetDefaultInputDeviceID()` and `Pa_GetDefaultOutputDeviceID()` functions may be used to retrieve the ids of the host's preferred input and output devices.

3.2. Stream Management

PortAudio streams may be opened with either the `Pa_OpenStream()` or `Pa_OpenDefaultStream()` functions, both of which return an opaque handle to a PortAudioStream object.

`Pa_OpenStream()` allows specification of: input and output devices; sample format; number of channels and a device specific information block for input and output; sample rate; number and size of i/o buffers; a set of implementation defined stream flags; the user callback function and a user specified data pointer which is passed to the callback function.

`Pa_OpenDefaultStream()` provides a simplified interface for opening a stream using the default device(s)

The `Pa_StartStream()` and `Pa_StopStream()` functions are used to begin and end processing on a stream. `Pa_AbortStream()` may be used to immediately abort playback on a stream rather than waiting for queued samples to finish playing.

When the stream is started, audio I/O begins and the user callback function is repeatedly called with a pointer to a full input buffer that is to be read, and a pointer to an empty output buffer that is to be filled. It is also passed a time-stamp that is the number of sample frames generated so far. The time-stamp of the currently playing sample can also be queried which allows the audio output to be synchronized with MIDI or video events.

The user defined callback function supplied to `Pa_OpenStream()` and `Pa_OpenDefaultStream()` has the following prototype:

```
typedef int (PortAudioCallback)(
        void *inputBuffer,
        void *outputBuffer,
        unsigned long framesPerBuffer,
        PaTimestamp outTime,
        void *userData );
```

The callback function may return a non-zero value to indicate that use of the stream has completed. The main program can determine whether a stream has completed using the `Pa_StreamActive()` function.

`Pa_StreamTime()` returns the current playback time of a stream. It is intended for use as a time reference when synchronising audio to MIDI.

`Pa_GetCPULoad()` returns a floating point value ranging from zero to one which indicates the fraction of total CPU time being consumed by the stream's audio processing. This gives the programmer an opportunity to modify their synthesis techniques, or to reduce the number of voices, in order to prevent excessive loading of the CPU.

The `Pa_CloseStream()` function should be used to close a stream when it is no longer needed.

3.3. Error Handling

Most PortAudio functions return an error code of type PaError. The `Pa_GetErrorText(err)` function may be used to retrieve textual error information. PortAudio also provides the `Pa_GetHostError()` function for retrieving host specific error codes.

4. Existing Implementations

At the time of writing, stable PortAudio implementations exist for Windows (WMME and DirectX), the Apple Macintosh Sound Manager, and for Unix using the OSS driver model.

The functions common to all platforms are implemented in a shared module. These include error code interpretation, parameter checking, clipping, dithering, and conversion between different data formats. Implementation for a new platform, therefore, only involves the implementation of host specific functions. An implementation guide is provided as well as a programming tutorial. There is also a suite of test programs and example programs that can be used to validate a new implementation.

The **Windows Multimedia API** implementation utilises a high priority worker thread to process sample buffers. A Win32 Event synchronization object is used to allow the soundcard driver to efficiently wake the worker thread when an input buffer is full or an output buffer is empty. When the worker thread wakes, it gathers the available buffers and passess them to the user defined callback function. It calls `waveOutWrite()` to output the data generated by the callback function, and requeues used input buffers by calling

waveInAddBuffer()

The **DirectSound** implementation was originally written using a background thread that waited for notification events from DirectSound. But using this technique resulted in high latency so the implementation was changed to use a periodic timer callback based on timeSetEvent(). The timer callback function determines the amount of space available in the DirectSound ring buffers and fills them with data supplied by the user's PortAudio callback function.

The **Macintosh Sound Manager** implementation was originally written using the SndDoubleBufferPlayback() call. But that function is not supported in Mac OS X. So it was rewritten using SndDoCommand() for output and SPBRecord() for input, both of which work with Mac OS 7,8,9 and Mac OS X under Carbon. If both input and output are requested, then data collected from the SPBRecord() callback is gathered and written to an atomic ring buffer. The SndDoCommand() callback then reads the input data from the ring buffer and passes it to the PortAudio callback which generates the output data. This allows the user to process input data and output the result using a single callback function.

The **Unix** implementation spawns a thread that reads "/dev/audio", passes the data to the user callback, then writes the returned data to "/dev/audio". It uses OSS, or the OSS subset of the ALSA API. (OSS)

5. Design Considerations

The PortAudio API arose from discussions on the music-dsp mailing list during 1998. The following requirements influenced the design of the API:

5.1. Requirements

- Implementation should be possible on all common computer music platforms. In 1998 this included: Pre OS-X Macintosh systems, 32 bit Microsoft Windows systems, BeOS™ and various flavours of Unix™ including Irix™ and Linux. Implementation in embedded systems, and interfacing to proprietary APIs such as ASIO™, EASI™ and Re-Wire was also considered.
- Efficient, and ideally optimal use of the audio services on all target platforms should be possible.
- The API should be simple enough to be used by music students with minimal experience in C programming.
- The API should seek to provide only low-level audio services, and to only support those services directly available on the host platform. Emulation features such as sample rate conversion were considered to be beyond the scope of the API.

5.2. Concurrency

A significant constraint imposed by the range of proposed target platforms is the variety of concurrency mechanisms supported. For example the Macintosh provides no way of implementing mutual exclusion between an interrupt level audio processing callback and the main program.

An early version of the API provided Lock and Unlock methods to implement mutual exclusion. These methods were later removed, as they could not be implemented on all target platforms. It is recommended that applications use atomic data structures such as FIFOs for communication between the main program and the audio processing callback. This technique is used in JSyn, and other applications, and has proven to be both robust and portable. An atomic FIFO implementation in 'C' is supplied in a utility library.

5.3. Callbacks versus Blocking I/O

The programmer must supply a pointer to a user callback function. We decided to use a callback function instead of a blocking read/write interface because a callback function provides a way to do audio synthesis in a background "thread". This is important because there is no cross platform mechanism for launching a thread in C, and also because the Macintosh (prior to Mac OS X) does not have strong support for multi-tasking. It was also considered likely that any large application would require a background thread for audio even if blocking I/O were available.

The authors believe that callbacks are the preferred technique for portable APIs. In order, however, to support the development of extremely simple applications for pedagogical purposes, we have provided a blocking I/O utility layer on top of the PortAudio API.

5.4. Latency

PortAudio acts as a software layer that maps between the PortAudio API and the underlying host dependant API. As such, it generally does not add any latency to what an application would have by interfacing directly to the underlying API. The programmer can determine the minimum latency by passing the buffer size and the number of buffers to Pa_OpenStream(). Or it can simply specify a small buffer size and pass zero for the number of buffers. PortAudio will then use the minimum number of buffers determined to ensure reliable operation. On some platforms, such as Windows and Linux, the user may specify the minimum system latency by setting an environment variable that is read by PortAudio.

6. Future Work

The proliferation of alternative APIs and driver models for real-time audio transport on some platforms (MME, Direct-X and ASIO all being popular on Windows for example) necessitates support for multiple driver models on a single platform. It has been proposed that the PortAudio implementation should virtualize support for multiple host APIs. This is, however, unlikely to affect the PortAudio API itself.

We are also finishing up a Mac Core Audio implementation, and are planning implementations for various plugin APIs such as VST.

We are also considering adding low-level support for blocking I/O to support applications where the audio is generated in the foreground process.

7. References

1991, "Sound, Music, and Signal Processing on a NeXT Computer: Concepts", NeXT Computer, Inc. Addison Wesley Publishing Company.

1993, "Digital Media Programming Guide: Audio, MIDI and Compression", Silicon Graphics Inc.

8. Links

Apple Macintosh Sound Manager,
`http://developer.apple.com/audio/`

Microsoft DirectX Audio API and Windows MultiMedia Extensions (WMME),
`http://msdn.microsoft.com/library/`

The music-dsp mailing list,
`shoko.calarts.edu/~glmrboy/musicdsp/`

OSS - OpenSound API,
`http://www.opensound.com/`

PortMusic Project,
`http://www.cs.cmu.edu/~music/portmusic/`

Listing 1, Minimal PortAudio Program to perform Ring Modulation and Mixing

```
#include "stdio.h"
#include "portaudio.h"
/* This will be called asynchronously by the PortAudio engine. */
static int myCallback( void *inputBuffer, void *outputBuffer,
        unsigned long framesPerBuffer, PaTimestamp outTime, void *userData )
{
    float *out = (float *) outputBuffer;
    float *in  = (float *) inputBuffer;
    float leftInput, rightInput;
    unsigned int i;
    if( inputBuffer == NULL ) return 0;
/* Read input buffer, process data, and fill output buffer. */
    for( i=0; i<framesPerBuffer; i++ )
    {
        leftInput = *in++;      /* Get interleaved samples from input buffer. */
        rightInput = *in++;
        *out++ = leftInput * rightInput;            /* ring modulation */
        *out++ = 0.5f * (leftInput + rightInput);   /* mixing */
    }
    return 0;
}
/* Use a PortAudioStream to process audio data. */
int main(void)
{
    PortAudioStream *stream;
    Pa_Initialize();
    Pa_OpenDefaultStream(
        &stream,
        2, 2,           /* stereo input and output */
        paFloat32,  44100.0,
        64,  0,         /* 64 frames per buffer, let PA determine numBuffers */
        myCallback, NULL );
    Pa_StartStream( stream );
    Pa_Sleep( 10000 );    /* Sleep for 10 seconds while processing. */
    Pa_StopStream( stream );
    Pa_CloseStream( stream );
    Pa_Terminate();
    return 0;
}
```