GRAME - Computer Music Research Lab.
Technical Note - 01-11-06

Porting PortAudio API on ASIO

Stéphane Letz
November 2001
Grame - Computer Music Research Laboratory
9, rue du Garet BP 1185 69202 FR - LYON Cedex 01
letz@grame.fr

Abstract

This document describes a port of the PortAudio API using the ASIO API on Macintosh and Windows. It explains technical choices used, buffer size adaptation techniques that guarantee minimal additional latency, results and limitations.

1 The ASIO API

ASIO (Audio Streaming Input Ouput) is an API defined and proposed by Steinberg. It addesses the area of efficient audio processing, synchronization, low latency and extentibility on the hardware side. ASIO allows the handling of multi-channel professional audio cards, and different sample rates (32 kHz to 96 kHz), different sample formats (16, 24, 32 bits of 32/64 floating point formats). ASIO is available on MacOS and Windows.

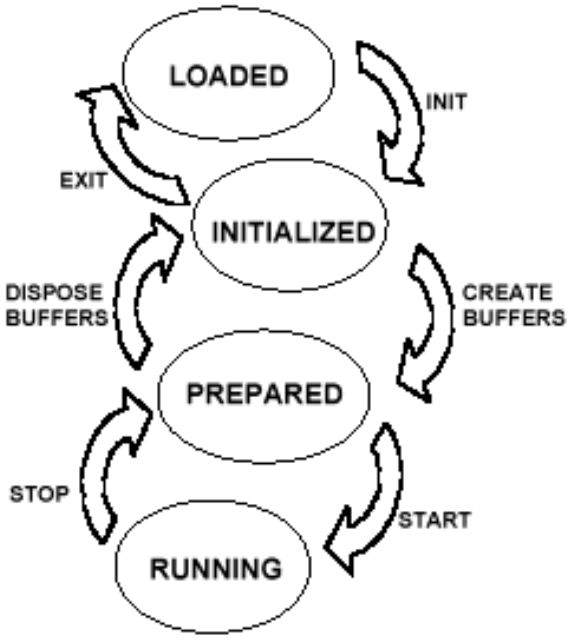
2 PortAudio API

PortAudio is a library that provides streaming audio input and output. It is a cross-platform API that works on Windows, Macintosh, Linux, SGI, FreeBSD and BeOS. This means that programs that need to process or generate an audio signal, can run on several different computers just by recompiling the source code. PortAudio is intended to promote the exchange of audio synthesis software between developers on different platforms.

3 Technical choices

Porting PortAudio to ASIO means that some technical choices have to be made. The life cycle of the ASIO driver must be "mapped" to the life cycle of a PortAudio application. Each PortAudio function will be implemented using one or more ASIO functions.

3.1 Life cycle of the ASIO driver :



The most important functions that must be used to open and use the ASIO driver are the following :

- From **Loaded** to **Initialized** state : **ASIOInit**
- From **Initialized** to **Prepared** state : **ASIOCreateBuffers**
- From **Prepared** to **Running** state : **ASIOStart**

ASIO drivers are callback based. In their running state, an audio callback will be called with the index of the double buffer half (0 or 1) which has to be processed by the application.

The most important functions that must be used to close the ASIO driver are the following :

- From **Running** to **Prepared** state **: ASIOStop**
- From **Prepared** to **Initialized** state **: ASIODisposeBuffers**
- From **Initialized** to **Loaded** state **: ASIOExit**

3.2 Life cycle of an PortAudio application

A standard PortAudio application uses the following functions during it's life time :

- **Pa_Initialize** : to initialize the library
- **Pa_OpenSteam** : to access the diver and open a stream
- **Pa_StartStream** : to start audio processing
- **Pa_StopStream** : to stop audio processing
- **Pa_CloseStream** : to deallocate the stream structure
- **Pa_Terminate :** to deallocate the library structures

3.3 PortAudio to ASIO mapping

Most PortAudio API functions have an internal platform dependant version that must be implemented on each platform. They are named **PaHost_XXXX.** Aditional functions needed for the PortAudio implementation on ASIO are named **Pa_ASIO_XXX**. Each external PortAudio API function is implemented using a **PaHost_XXXX** function that may use other functions of the PortAudio API. **PaHost_XXXX** functions call one or several "glue" functions named **Pa_ASIO_XXX** that in turn call one or several ASIO API functions.
The following table describes the "mapping " between " PortAudio functions and ASIO functions :

External PortAudio API	Implementation PortAudio API	Glue	ASIO API
Pa_Initialize	PaHost_Init , Pa_CountDevice	Pa_ASIO_QueryDeviceInfo	**loadAsioDriver** **ASIOInit** **ASIOGetChannels** **ASIOCanSampleRate** **ASIOGetChannelInfo**
Pa_OpenSteam	PaHost_OpenStream	Pa_ASIO_loadDevice	**loadAsioDriver** **ASIOInit** **ASIOGetChannels** **ASIOGetBufferSize**
	PaHost_CalcNumHostBuffers		
		Pa_ASIO_CreateBuffers	**ASIOCreateBuffers** **ASIOGetLatencies** **ASIOSetSampleRate**

Pa_StartStream	PaHost_StartOutput PaHost_StartInput PaHost_StartEngine	**ASIOStart**
Pa_StopStream	PaHost_StopOutput PaHost_StopInput PaHost_StopEngine	**ASIOStop**
Pa_CloseStream	PaHost_CloseStream	**ASIODisposeBuffers** **ASIOExit**
Pa_Terminate	PaHost_Term	**removeCurrentDriver**

3.3.1 Opening : getting device information

On Macintosh ASIO drivers are files located in a special folder called “ASIO Drivers” located in the application folder. These ASIO drivers can easilly be changed simply by moving ASIO drivers files to and from the ASIO Drivers folder. Some utilities functions are available in the the ASIO SDK :

- **loadDriver** load an ASIO driver in memory given it’s name
- **getDriverNames** returns the name of all available drivers

The internal function **Pa_ASIO_QueryDeviceInfo** use them to get access to the available ASIO drivers on the machine.

3.3.2 Configuration

Each ASIO driver has to be loaded, configurated by choosing the number of input/ouput channels, and the size of ASIO internal buffers. Selection and configuration of the ASIO driver is done in **PaHost_OpenStream :**

- the ASIO driver corresponding to the selected device is loaded using **Pa_ASIO_loadDevice** which calls **ASIOInit**
- the ASIO driver sample rate is configurated using **ASIOSetSampleRate**
- the ASIO driver buffer size is computed using **PaHost_CalcNumHostBuffers**
- ASIO buffers, channels and callback are allocated and configured using **Pa_ASIO_CreateBuffers** which calls **ASIOCreateBuffers**

3.3.3 Start/Stop

The **PaHost_StartEngine** function directly calls the **ASIOStart** function to start the audio streaming process.
The **PaHost_StopEngine** function directly calls the **ASIOStop** function to stop the audio streaming process.

3.3.4 Closing

The **PaHost_CloseStream** function will call **ASIODisposeBuffers** then **ASIOExit.**

3.3.5 Sample count

ASIO gives the sample position in the audio stream at each callback. This information is directly used to update the **pahsc_NumFramesDone** variable that is returned by the PortAudio **Pa_StreamTime** function.

3.3.6 Audio callback

The ASIO driver calls a **bufferSwith** or **bufferSwitchTimeInfo** callback with the index of the double buffer half (0 or 1) which has to be processed, to the application. Upon return of the callback, the application has read all input data and provided all output data. In this document we will use the name **host buffers** for **ASIO internal buffers** and **user buffers** for **PortAudio buffers**.

After the initialisation steps, the ASIO callback will basically do the following operations in a full-duplex case :

• transfer samples from the **ASIO host input buffer** to the **PortAudio user input buffer**. This transfer implies interleaving and possible samples conversion in the current implementation.

• if the PortAudio user input buffer is full, call the PortAudio callback which will produce a user output buffer.

• transfer samples from the **PortAudio user ouput buffer** into the **ASIO host ouput buffer**. This transfer needs “de-interleaving” and possible sample conversion in the current implementation.

These operations may be done several time depending of the ASIO and PortAudio buffer sizes that are used. But at each ASIO callback, **all samples from the host input buffer must be consumed** and a **complete host ouput buffer has to be produced**.

4 Buffer size adaptation techniques

A PortAudio program opens a stream by defining a buffer size (expressed in frames) that the audio callback must receive with the **exact number** of required frames. On the driver side, ASIO gives 3 values : **minimum size, preferred size** and **maximum size** for its internal buffers. Some ASIO drivers give the same value for minimum, prefered and maximum buffer size and only an external tool can be used to change the host buffer size when the driver is not running (M Audio Delta 10x10 for example)

The main difference with other implementation or the PortAudio API is that ASIO imposes it’s buffer size, thus in the general case buffer size adaptation techniques have to be used. We want to use the minimum host buffer size especially in full-duplex applications when the minimal latency is often necessary.

4.1 Computing host buffer size

Knowing the required user buffer size and the number of buffers, the first step is to compute the host buffer size that is most adapted using the following method :

• computes the global **requested user buffer size = numBuffer * userBufferSize**

• if the global requested user size is inside the minimum/maximum host size range, take :
host buffer size = requested user buffer size

• otherwise

• if **requested user buffer size < miminum host buffer size** take the **first multiple of requested user size immediatly superior of miminum host buffer size** if possible
• otherwise take **host buffer size = miminum host buffer size**

• if **requested user buffer size > maximum host buffer size** take the **first divisor of requested user buffer size immediatly inferior of miminum host buffer size** if possible
• otherwise take **host buffer size = maximum host buffer size**

• align **host buffer size** to be a power of 2 if the ASIO drivers requires it

4.2 Buffer size adapdation techniques

A specific technique that allows to adapt two callback using buffers of different sizes has been developed. It is explained in detail in [Letz 2001].

5 Implementation

This general algorithm (finding the smallest number of frames for ouput shift) is actually usable in all cases : simple ones when M and N are multiple or divisors of each other and the more complex ones. In the implementation, several variables are used during the audio computation :

• **pahsc_userInputBufferFrameOffset** : position in input user buffer

• **pahsc_userOutputBufferFrameOffset** : position in output user buffer
• **pahsc_hostOutputBufferFrameOffset** : position in output ASIO buffer

At initialization time, a **Pa_ASIO_CalcFrameShift** function that implements the algorithm is used to compute the frame number used for the first host ouput buffer :

• when M>N we define **pahsc_OutputBufferOffset** = **Pa_ASIO_CalcFrameShift(M,N).** This value is used to shift the **pahsc_hostOutputBufferFrameOffset** variable (host ouput write offset)

• when M<N we define **pahsc_InputBufferOffset** = **Pa_ASIO_CalcFrameShift(M,N).** This value is used to shift the **pahsc_userInputBufferFrameOffset** variable (user input write offset)

5.1 Total Latency

After host buffer creation the **ASIOGetLatencies** function can be used to know the **input and ouput latencies.** Depending on the drivers internal implementation (if it writes directly to a DMA buffer or not) ouput latency will be 1 or 2 blocks. The total I/O latency will be :

ASIO input latency + ASIO output latency + host/user buffers size adaptation latency

5.2 Adtional function description

Here is the description of the most important functions of the PortAudio on ASIO implementation :

• **Pa_ASIO_QueryDeviceInfo** : load all available ASIO drivers and get information on all of them.

• **Pa_ASIO_loadDevice :** load the ASIO driver corresponding to the required device number.

• **Pa_ASIO_CreateBuffers** : create ASIO buffers and initialise ASIO channels.

• **Pa_ASIO_Convert_SampleFormat :** convert an ASIO native format sample into the corresponding PortAudio sample format.

• **Pa_ASIO_Convert_Inter_Input :** convert ASIO native buffers into PortAudio user buffers, do sample conversion and interleaving.

• **Pa_ASIO_Convert_Inter_Ouput :** convert PortAudio user buffers into ASIO native buffers, do sample conversion and “de-interleaving”.

• **Pa_ASIO_Callback_Input**, **Pa_ASIO_Callback_Output :** called by the native ASIO callback, do the buffer adpatation code, host to user buffer conversion, call of the PortAudio callback, production of the host ouput buffer.

• **PaHost_CalcNumHostBuffers** : from the **userBufferSize** and the **numOfBuffers** values, computes the most adpated hostBufferSize to be used for ASIO internal buffers creation.

• **Input_Int16_Flot32** and functions of the same family : host to user buffer conversion.

• **Output_Float32_Int16** and functions of the same family : user to host buffer conversion.

6 Limitations and possible improvements

6.1 Getting the smallest latency

To get minimal latencies in all cases, a better solution would be to allow the PortAudio user to know the host’s choosen buffer size. Thus, an application that wants small latency and does not care what the user buffer size’s can take the user buffer size defined in Pa_OpenStream to be exactly equal to the host buffer size.

6.2 Interleaved, non-interleaved mode.

The ASIO buffers are not interleaved. Because PortAudio currently use only interleaved buffers, “de-interleaving” has

to be done when converting ASIO buffers into user buffers, and the contrary at the ouput. This process is mixed with sample conversion.
A possible improvement is to extend PortAudio to allow the use of non-interleaved buffers. Thus de-interleaving is not longer necessary and fast vector operations can be used on some architectures (Intel SSE or PowerPC Altivec) to improve to speed of sample conversions.

6.3 Muliple buffers

When opening a PortAudio stream, a number of buffers can be specified. In the current implementation this number is used when computing the host buffer size. Specifying a small user buffer size and several buffers will result in a bigger host buffer size but that will always be limited by the maximum possible host buffer size.

6.4 Stream stopping

The current implementation does not distinguish between **aborting** and **stopping** the audio stream. Aborting should immediatly ends the stream and **stopping** should let the stream playing until the end of all buffers. The current implementation aborts the stream in all cases.

6.5 Handling multiple audio cards

ASIO does not allow to load and use several drivers at the same time. Thus only one audio stream can be opened using the currently loaded driver at a given time.

7 Conclusion

A port of the PortAudio API using the ASIO API on Macintosh and Windows has been done sucessfully, thus giving PortAudio based applications the possibility to be used with professional multi-channel cards on Macintosh and Windows. The described implementation handles the complete PortAudio API with some limitations that may be removed by extending the PortAudio API. Thanks to Phil Burk for his help and advice during this work, and to Nick Dodkovsky, David Viens and Maurice Cameron for testing on Windows.

References

[ASIO 97-99] Asio 2.0 Audio Streaming Input Ouput Developement kit. Steinberg (c) 1997-1999 Steinberg Soft - und harware GmbH.

[Bencina, Burk 2001] Ross bencina, Phil Burk “PortAudio : an Open Source Cross Platform Audio API” Proceedings of the International Computer Music Conference 2001, International Computer Music Association, San Francisco.

[Letz 2001] Stephane Letz “Callback adapdation techniques” GRAME - Computer Music Research Lab. Technical Note - 01-11-07