Open AL EAX-AC3 Extension

[Blank Page]

Contents

Introduction 4
EAX-AC3 Open AL Extension 5
Open AL Extensions 6
EAX-AC3 API Structures 7
alEAXAC3GetFunctionTable..8
EAXAC3QueryNumberOfDevices..10
EAXAC3QueryFile..11
EAXAC3QueryMemory 12
EAXAC3QueryDeviceCaps 13
EAXAC3OpenPlaybackDevice 14
EAXAC3ClosePlaybackDevice...15
EAXAC3OpenStream 16
EAXAC3CloseStream..18
EAXAC3PlayStream..19
EAXAC3PrePlayStream 20
EAXAC3QueryNoOfFramesReqForPlayback 21
EAXAC3StopStream 22
EAXAC3PauseStream 23
EAXAC3ResumeStream...24
EAXAC3SetFilePosition...25
EAXAC3GetPosition 26
EAXAC3LockBuffer 27
EAXAC3UnLockBuffer..29
EAXAC3GetErrorString 30
EAXAC3GetLastError 31
EAXAC3SetPlaybackMode 32
Code Examples 33

Introduction

The purpose of the EAX-AC3 extension to Open AL is to provide a convenient, high-level way for programmers to play back Dolby Digital content. This version of the extension supports Creative's SB Live! series of soundcards. The newer SB Live! 5.1 soundcards can decode a Dolby Digital stream, mix it with other playing audio streams, and output it via the analogue outputs. On the legacy SB Live! soundcards, which do not ship with Dolby Digital decoders, the Dolby Digital stream will be sent directly to the S/PDIF output for connection to an external Dolby Digital decoder. Please note that it is possible for an end-user to configure their SB Live! 5.1 card to send the Dolby Digital stream to the S/PDIF Output rather than decode it on-board. This option appears under 'Advanced' in the Speaker Control panel.

Currently a software based Dolby Digital decoder is not included with the Open AL extension. It is therefore recommended that you provide a 2-channel down-mixed version of your 5.1 tracks (possibly encoded in Dolby Surround) for those users without Dolby Digital decoding capabilities.

This API supports all of the different Dolby Digital bit-rates, however the audio must be encoded at 48KHz.

Dolby Digital content can exist either as part of a full-motion video (FMV) sequence, or as stand-alone music / atmospheric ambience to be played during game play. In this second scenario it will be necessary to mix the Dolby Digital audio with the sound effects generated by the game.

In order to provide the end-user with the best experience, it is necessary to be aware of the different soundcard / speaker configurations that exist, and how they will affect playback of Dolby Digital.

There are three different scenarios to be aware of :-

1. SB Live! 5.1 with multi-speaker system (such as FPS2000, or FPS2200) and AC3 Decode switched 'on' in the speaker control panel.

In this scenario the AC3 data will be decoded by the soundcard and mixed with other streams of audio data, before being sent to the users speaker system. This is the ideal delivery platform, which will enable the end-user to experience all of the audio. (If the user has less than 6 speakers, then the AC3 data will be decoded to the appropriate number of channels.)

Dolby Digital in FMV	OK
Dolby Digital in-game	OK

2. SB Live! 5.1 with Dolby Digital decoding multi-speaker system (such as DTT3500) and AC3 Decode switched 'off' in the speaker control panel.

In this scenario, the end-user has selected to use an external Dolby Digital decoder for AC3 decoding. The soundcard will direct the AC3 data directly to the S/PDIF Out, and any other audio to the analogue outputs.

In this scenario the end-user will have to mix the 6 channels of externally decoded Dolby Digital sound, with the (up to 6) analogue outputs of the soundcard.

Dolby Digital in FMV	OK (using external decoder)
Dolby Digital in-game	Requires external Mixer

3. SB Live! with multi-speaker system

In this final scenario, the AC3 data will always be sent directly to the S/PDIF output, and all other audio to the analogue outputs.

Dolby Digital in FMV	Requires external Dolby Digital decoder
Dolby Digital in-game	Requires external Dolby Digital decoder and Mixer

It is not possible to detect the presence of an external Dolby Digital decoder, so it is recommended that you ask the end-user if they have either a SB Live! 5.1 card, or an external Dolby Digital decoder connected to the S/PDIF Output of their SB Live!

To help confirm if the end-user has set-up their system correctly, it may be worth considering playing a test AC3 track and confirming that the end-user can hear it. If you are planning to play Dolby Digital tracks during game play, then extend this test to play Dolby Digital together with some sound effects, and ask the end-user if they can hear both.

EAX-AC3 Open AL Extension

This API provides three alternative ways to playback a Dolby Digital stream. If your Dolby Digital content is stored as a .ac3 file, then the DLL can take care of streaming the data directly from disk for you. Alternatively if your content is stored in memory, a proprietary format, or as part of an interleaved video file, then you can pass the DLL chunks of AC3 data at regular intervals. This can be achieved via a 'push' mechanism whereby your application will continuously supply the DLL with AC3 data when necessary, or via a 'Pull' mechanism whereby the DLL (via a Callback function) will notify your application when more AC3 data is required.

When using the DLL in 'push' or 'pull' mode, a conceptually circular AC3 data buffer will be created. At appropriate intervals the calling application will Lock a part of this buffer and write new AC3 data into it.

In all cases, the AC3 data will be packaged into the appropriate format to be sent to the Wave Device using the Multimedia Sound System (MMSYSTEM). Depending upon whether the soundcard includes a Dolby Digital decoder (see above) this AC3 data will either be decoded by the soundcard, or sent to the S/PDIF Output for connection to an external Dolby Digital decoder.

This version of the EAX-AC3 API supports Microsoft Windows 95, 98, and Millennium. Dolby Digital pass-thru is currently not supported in Windows 2000.

Open AL Extensions

Open AL provides extensions as a way to allow future growth of the API. The way to use an extension is specified in the Open AL documentation. The basic steps involved are querying for the extension, and then obtaining the address of an extension entry point.

To make it is quicker and easier to use the EAX-AC3 extension, you can only obtain the address of one function – *alEAXAC3GetFunctionTable()*. This function will take the address of an EAXAC3FNTABLE structure, and fill it with the addresses of all EAX-AC3 functions. The calling application can then access all of the functions in this API via the appropriate entry in the function table.

The eaxac3.h file includes a type definition of a pointer to the above function, which should be used to cast the return type of the *alGetProcAddress()* function to.

```
typedef ALboolean (*LPALEAXAC3GETFUNCTIONTABLE)    (LPEAXAC3FNTABLE);
```

EAX-AC3 API Structures

EAXAC3DEVICEINFO

```
typedef struct
{
      char szDeviceName[256];        // AC3 Device Name
      unsigned int uFlags;           // Flags specifying output types (see below)
      unsigned int uStreams;         // Number of AC3 Streams supported
      unsigned int uReserved;        // Reserved
} EAXAC3DEVICEINFO, *LPEAXAC3DEVICEINFO;
```

The following flags have been defined :-

```
UNKNOWN            // Output type(s) cannot be determined
SPDIFPASSTHRU      // EAX-AC3 Device supports S/PDIF pass-thru
FULLDECODE         // EAX-AC3 Device can decode Dolby Digital
```

AC3FILEINFO

```
typedef struct
{
      unsigned int nNumOfAC3Frames;      // Number of AC3 Frames in data
      unsigned int nAC3FrameSize;        // AC3 Frame size
      unsigned int nSizeOfFile;          // Size of file in bytes
      unsigned int nDuration;            // Time to play AC3 stream in milliseconds
      unsigned int nFrequency;           // Frequency of AC3 data
} AC3FILEINFO;
```

alEAXAC3GetFunctionTable

ALboolean alEAXAC3GetFunctionTable(LPEAXAC3FNTABLE *lpEAXAC3FnTable*)

Description

To use the features of the EAX-AC3 Open AL extension, an application should (after verifying that the EAX-AC3 extension is present (see above)) obtain the address of this function.

This function will take the address of an EAXAC3FNTABLE structure, and fill it with the addresses of all the functions contained in the EAX-AC3 API.

```
LPEAXAC3FNTABLE     g_lpEAXAC3FnTable;

…


LPALEAXAC3GETFUNCTIONTABLE ALEAXAC3GetFunctionTable;

// Retrieve the address of the alEAXAC3GetFunctionTable () function

ALubyte szFnName[] = "alEAXAC3GetFunctionTable";
ALEAXAC3GetFunctionTable = (LPALEAXAC3GETFUNCTIONTABLE) alGetProcAddress (szFnName);

if (ALEAXAC3GetFunctionTable == NULL)
{
      printf ("Cannot find alEAXAC3GetFunctionTable function\n");
      return 0;
}

// Use the ALEAXAC3GetFunctionTable () function to retrieve a function table containing all of the
// EAX-AC3 Functions

// First allocate memory for the Function Table
g_lpEAXAC3FnTable = new EAXAC3FNTABLE;

if (g_lpEAXAC3FnTable == NULL)
      return 0;

if (!ALEAXAC3GetFunctionTable(g_lpEAXAC3FnTable))
{
      printf ("Failed to retrieve the EAX-AC3 Function Table\n");
      return 0;
}

// Find out how many AC3 Devices are available
nNumberOfEAXAC3Devices = g_lpEAXAC3FnTable->EAXAC3QueryNumberOfDevices();

…
```

Parameters

lpEAXAC3FnTable (IN)

Address of an EAXAC3FNTABLE structure. This function will fill the structure with the addresses of all the functions in the EAX-AC3 API. The application is responsible for allocating (and deleting) the memory for the EAX-AC3 function table.

Returns

AL_TRUE if successful
AL_FALSE if not

EAXAC3QueryNumberOfDevices

int EAXAC3QueryNumberOfDevices ()

Description

Function will return the number of EAX-AC3 Devices available for use on the system.

Returns

The number of EAX-AC3 devices on the system.

EAXAC3QueryFile

HRESULT EAXAC3QueryFile(*char *szAC3Filename*, AC3FILEINFO **lpAC3FileInfo*, int *nSizeOfAC3FileInfoStruct*)

Description

This function will examine the AC3 File supplied and report some of its statistics. As this function will be stepping through the entire AC3 file it is best not to use this function at time critical stages, especially on long AC3 files.

Parameters

szAC3Filename (IN)

Full filename (including path) of the AC3 file to be examined

lpAC3FileInfo (OUT)

Address of an AC3FILEINFO structure to be filled with information about the AC3 stream.

nSizeOfAC3FileInfoStruct (IN)

Size of the AC3FILEINFO structure pointed to be *lpAC3FileInfo*.

Returns

EAXAC3_OK if successful
EAXAC3ERR_FILENOTFOUND if the file is not found
EAXAC3ERR_AC3FILETOBIG if the AC3 file is larger than 2^32 bytes
EAXAC3ERR_INVALIDAC3FRAME if an invalid AC3 frame is found
EAXAC3ERR_AC3NOTAT48KHZ if the AC3 data hasn't been encoded at 48KHz
EAXAC3ERR_INVALIDPARAMETER if *nSizeOfAC3FileInfoStruct* is not a valid size

EAXAC3QueryMemory

HRESULT EAXAC3QueryMemory (char **lpBuffer*, int *nSizeOfBuffer*, AC3FILEINFO **lpAC3FileInfo*, int *nSizeOfAC3FileInfoStruct*)

Description

Examines the AC3 data stored at the given address and reports some statistics about it.

Parameters

lpBuffer (IN)

Pointer to memory location containing the AC3 Data

nSizeOfBuffer (IN)

Size of the data pointed to by *lpBuffer*

lpAC3FileInfo (OUT)

Address of an AC3FILEINFO structure to be filled with information about the AC3 data.

nSizeOfAC3FileInfoStruct (IN)

Size of the AC3FILEINFO structure pointed to by *lpAC3FileInfo*.

Returns

EAXAC3_OK when completed
EAXAC3ERR_NOTENOUGHDATA if there is not enough AC3 data to calculate AC3 statistics
EAXAC3ERR_INVALIDAC3FRAME if an invalid AC3 frame is discovered
EAXAC3ERR_AC3NOTAT48KHZ if the AC3 data has not been encoded at 48KHz
EAXAC3ERR_INVALIDPARAMETER if *nSizeOfAC3FileInfoStruct* is not a valid size

EAXAC3QueryDeviceCaps

HRESULT EAXAC3QueryDeviceCaps (EAXAC3HANDLE *EAXAC3Handle*, LPEAXAC3DEVICEINFO *lpEAXAC3DeviceInfo*, int *nSizeOfEAXAC3DeviceInfoStruct*)

Description

Retrieve the capabilities of the given EAX-AC3 Device.

Parameters

EAXAC3Handle (IN)

EAX-AC3 device number – valid range is 0 to one less than the number of EAX-AC3 devices. Use *EAXAC3QueryNumberOfDevices* to find out how many EAX-AC3 devices are available on the end-users system.

lpEAXAC3DeviceInfo (OUT)

Address of the EAXAC3DEVICEINFO structure to be filled with information about the EAX-AC3 device.

nSizeOfEAXAC3DeviceInfoStruct (IN)

Size of the EAXAC3DEVICEINFO structure pointed to by *lpEAXAC3DeviceInfo*.

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Playback device has not been opened
EAXAC3ERR_WAVEOUTERROR if there is an error with the Wave Out Device

EAXAC3OpenPlaybackDevice

HRESULT EAXAC3OpenPlaybackDevice(EAXAC3HANDLE *EAXAC3Handle*)

Description

This function attempts to open the selected EAX-AC3 device ready for playback of Dolby Digital content.

Parameters

EAXAC3Handle (IN)

Specifies the EAX-AC3 device number to open. This number will be in the range of 0 to one less than the number of EAX-AC3 devices on the system.

Returns

EAXAC3_OK if successful
EAXAC3ERR_UNABLETOOPENEAXAC3DEVICE if the EAX-AC3 device cannot be used
EAXAC3ERR_OUTOFMEMORY if not enough memory can be allocated

EAXAC3ClosePlaybackDevice

HRESULT EAXAC3ClosePlaybackDevice(EAXAC3HANDLE *EAXAC3Handle*)

Description

Closes the given EAX-AC3 playback device. Any playing AC3 streams will be stopped and closed down.

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Playback device has not been opened
EAXAC3ERR_WAVEOUTERROR if there is an error with the Wave Out Device

EAXAC3OpenStream

HRESULT EAXAC3OpenStream(EAXAC3HANDLE *EAXAC3Handle*, AC3STREAM **lpAC3Stream*, LPAC3CALLBACK *pAC3CallbackFn*, char **szAC3Filename*, SOURCE *src*);

Description

Opens an AC3 Stream ready for playback on the selected EAX-AC3 device. A handle to the opened AC3 stream is returned in the second parameter. If desired, the calling application can provide an address of a *callback* function to receive messages about playback status. The source of the stream can be from file (specified in *szAC3Filename*) or from memory. If the source is memory, then you must set *szAC3Filename* to either "PUSH" or "PULL" depending upon how you would like to send AC3 data to the EAX-AC3 device.

In 'PUSH' mode – the only time that audio data is sent to the EAX-AC3 device is when the calling application calls *EAXAC3UnlockBuffer*() to commit new AC3 data written into the AC3 Data buffer. The only *callback* message generated is EAXAC3REACHEDEND, which indicates that playback, has completed. For best performance it is recommended to write small amounts of AC3 data to the AC3 data buffer regularly.

In 'PULL' mode – the calling application is notified when more AC3 data is required via the EAXAC3NEEDMOREDATA *callback* message. Once this data has been written to the AC3 Data buffer, it is sent to the EAX-AC3 device.

Parameters

EAXAC3Handle (IN)

EAX-AC3 Device identifier

lpAC3Stream (OUT)

Receives a unique identifier for this particular AC3 Stream.

pAC3CallbackFN (IN)

Specified the address of a *callback* function, which will receive messages about playback status. The callback function should of type LPAC3CALLBACK, which is defined as :-

```
typedef void (__stdcall *LPAC3CALLBACK)(AC3STREAM AC3Stream, int msg);
```

The following *callback* messages have been defined :-

```
EAXAC3NEEDMOREDATA      // Sent when more AC3 data is required
EAXAC3REACHEDEND        // Sent when AC3 playback has completed
```

szAC3Filename (IN)

Name of the .ac3 file to stream from disk, or either "PUSH" or “PULL” if streaming the AC3 data from memory.

src (IN)

Source of the AC3 Stream - AC3FILE or MEMORY

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Playback device has not been opened
EAXAC3ERR_AC3STREAMALREADYOPEN if an AC3 stream is already open
EAXAC3ERR_OUTOFMEMORY if a request to allocate memory failed
EAXAC3ERR_FILENOTFOUND if the supplied AC3 File cannot be found
EAXAC3ERR_AC3FILETOBIG if the supplied AC3 File is larger than 2^32 bytes
EAXAC3ERR_AC3FRAMENOTFOUND if the first AC3 Frame cannot be found
EAXAC3ERR_AC3NOTAT48KHZ if the AC3 File has not been encoded at 48KHz

EAXAC3CloseStream

HRESULT EAXAC3CloseStream(AC3STREAM *AC3Stream*)

Description

Closes the selected AC3 Stream.

Parameters

AC3Stream (IN)

Identifier of AC3 stream to close.

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Playback device has not been opened
EAXAC3ERR_AC3STREAMNOTOPEN if an AC3 Stream has not been opened
EAXAC3ERR_WAVEOUTERROR if an error with the Wave Out device occurred

EAXAC3PlayStream

HRESULT EAXAC3PlayStream(AC3STREAM *AC3Stream*, bool *bLooping*)

Description

Begins playback of the selected AC3 Stream. The second parameter indicates whether playback should loop (this flag will only affect playback when using the API to playback a .ac3 file from disc (SOURCE == AC3FILE).

If playing from memory, it is necessary to have filled the AC3 data buffer with sufficient data to create a reasonable buffer of data for the Wave Out device. This amount has been currently been set to 16 AC3 Frames worth of data (equates to 512 milliseconds of uncompressed audio data). However, the application should call *EAXAC3QueryNoOfFramesReqForPlayback*() to determine this value. To calculate how much data this is for a particular AC3 source, use the appropriate *EAXAC3QueryFile()* or *EAXAC3QueryMemory()* function to return the size of an individual AC3 Frame.

Parameters

AC3Stream (IN)

AC3 Stream identifier.

bLooping (IN)

Boolean variable to determine if playback of the .ac3 file should be looped or not.

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Playback device has not been opened
EAXAC3ERR_AC3STREAMNOTOPEN if an AC3 Stream has not been opened
EAXAC3ERR_ALREADYPLAYING if an AC3 Stream is already playing
EAXAC3ERR_NOTENOUGHAC3DATAINAC3DATABUFFER if there isn't enough data to start playback
EAXAC3ERR_WAVEOUTPREPAREHEADERFAILURE if the Wave headers cannot be prepared
EAXAC3ERR_FAILEDTOCREATEEVENT if a Win 32 Event couldn't be created
EAXAC3ERR_WAVEOUTERROR if there is an error with the Wave Out Device

EAXAC3PrePlayStream

HRESULT EAXAC3PrePlayStream(AC3STREAM *AC3Stream*)

Description

This function will perform exactly the same operations as the *EAXAC3PlayStream* function – except for actually starting to play the data. There is a variable amount of overhead for setting up the buffers of AC3 data and sending them to the wave device. Therefore an application can call this function to perform all of the necessary initialisation procedures first, and then call the *EAXAC3PlayStream* function when ready to actually begin playback. This function should be used when an application requires the smallest time delay between calling *EAXAC3PlayStream* and actually hearing audio.

Parameters

AC3Stream (IN)

AC3 Stream identifier.

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Playback device has not been opened
EAXAC3ERR_AC3STREAMNOTOPEN if an AC3 Stream has not been opened
EAXAC3ERR_ALREADYPLAYING if an AC3 Stream is already playing
EAXAC3ERR_NOTENOUGHAC3DATAINAC3DATABUFFER if there isn't enough data to start playback
EAXAC3ERR_WAVEOUTPREPAREHEADERFAILURE if the Wave headers cannot be prepared
EAXAC3ERR_FAILEDTOCREATEEVENT if a Win 32 Event couldn't be created
EAXAC3ERR_WAVEOUTERROR if there is an error with the Wave Out Device

EAXAC3QueryNoOfFramesReqForPlayback

int EAXAC3QueryNoOfFramesReqForPlayback(AC3STREAM *AC3Stream*)

Description

This function will return the minimum number of AC3 Frames required in the AC3 Data buffer before Playback will commence. This is the minimum amount of data required to fill 2 Wave Out buffers.

Parameters

AC3Stream (IN)

AC3 Stream identifier.

Returns

The number of AC3 Frames required before playback will commence.

EAXAC3StopStream

HRESULT EAXAC3StopStream(AC3STREAM *AC3Stream*)

Description

Immediately stops playback of the appropriate AC3 stream.

Parameters

AC3Stream (IN)

AC3 Stream identifier.

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Device has not been opened
EAXAC3ERR_AC3STREAMNOTOPEN if an AC3 stream has not been opened
EAXAC3ERR_WAVEOUTERROR if there is a Wave Out error

EAXAC3PauseStream

HRESULT EAXAC3PauseStream(AC3STREAM *AC3Stream*)

Description

Pauses a playing AC3 stream.

Parameters

AC3Stream (IN)

AC3 Stream identifier.

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Device has not been opened
EAXAC3ERR_AC3STREAMNOTOPEN if an AC3 stream has not been opened
EAXAC3ERR_WAVEOUTERROR if there is a Wave Out error

EAXAC3ResumeStream

HRESULT EAXAC3ResumeStream(AC3STREAM *AC3Stream*)

Description

Resumes playback of a paused AC3 stream.

Parameters

AC3Stream (IN)

AC3 Stream identifier.

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Device has not been opened
EAXAC3ERR_AC3STREAMNOTOPEN if an AC3 stream has not been opened
EAXAC3ERR_WAVEOUTERROR if there is a Wave Out error

EAXAC3SetFilePosition

HRESULT EAXAC3SetFilePosition(AC3STREAM *AC3Stream*, POSFORMAT *posFormat*, int *nAmount*);

Description

Sets the read position in the AC3 File. This function has no effect if the AC3 Stream source is from memory. The new position in the file must point to the beginning of an AC3 synchronization frame. (If the desired position is in milliseconds, it will automatically be rounded up to the next multiple of 32ms).

If the stream is currently playing then it will be temporarily paused while the file pointer is moved to the new location. Any data already sent to the wave device will be invalidated.

Parameters

AC3Stream (IN)

AC3 Stream identifier.

posFormat (IN)

Enumerated variable indicating preference for position format (MILLISECONDS, BYTES or AC3FRAMES)

nAmount (IN)

The desired position in the file to move to, in the format supplied

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Playback device has not been opened
EAXAC3ERR_AC3STREAMNOTOPEN if an AC3 Stream has not been opened
EAXAC3ERR_SETPOSITIONONLYWORKSONAC3FILES if the AC3 stream is not from file
EAXAC3ERR_WAVEOUTERROR if there is an error with the Wave Out Device
EAXAC3ERR_POSITIONOUTOFRANGE if requested position in not within the file
EAXAC3ERR_NOTATSTARTOFAC3FRAME if requested position does not point to start of an AC3 Frame
EAXAC3ERR_INVALIDAC3FRAME if an invalid AC3 Frame is found

EAXAC3GetPosition

HRESULT EAXAC3GetPosition(AC3STREAM *AC3Stream*, POSFORMAT *posFormat*, int **lpAmount*)

Description

Retrieves the current play position of the AC3 Wave Stream. The position is relative to the start of the AC3 data sent to the Wave device. This value is reset when *EAXAC3StopStream* is called.

Like the *EAXAC3SetFilePosition* function, you can pass a preferred format for the position to be returned in (milliseconds, bytes or AC3 Frames).

Parameters

posFormat (IN)

Enumerated variable indicating preference for position type (MILLISECONDS, BYTES or AC3FRAMES)

lpAmount (OUT)

Returns the position in the format requested

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Playback device has not been opened
EAXAC3ERR_AC3STREAMNOTOPEN if an AC3 Stream has not been opened
EAXAC3ERR_WAVEOUTERROR if there is an error with the Wave Out Device

EAXAC3LockBuffer

HRESULT EAXAC3LockBuffer(AC3STREAM *AC3Stream*, unsigned long *ulBytes*, void ***ppvPointer1*, unsigned long **pulBytes1*, void ***ppvPointer2*, unsigned long **pulBytes2*, unsigned long *ulFlags*);

Description

This function is used to Lock part, or all, of the AC3 Data buffer in order to fill it with new AC3 data ready for playback. The AC3 Data buffer is conceptually circular, therefore when you request a Lock you should check to see if one or both of the pointers are valid, and then write only as much data to each memory location as indicated in the two size parameters.

If the calling application has requested a Lock on more data than is currently available, then the function will simply Lock as much data as possible. It is the calling applications responsibility to check the two size parameters and to never write more data than indicated to the appropriate locations.

The final parameter *ulFlags* can be set to either FROMWRITECURSOR or ENTIREBUFFER. In the first case, the Locked memory will begin from the current position of the AC3 Data Buffer Write Cursor. In the second case, if the AC3 Stream is not playing then the entire buffer will be Locked, otherwise the function will Lock as much data as possible, beginning at the AC3 Data Buffer's Write Cursor.

Parameters

AC3Stream (IN)

AC3 Stream identifier.

ulBytes (IN)

Requested amount of data to Lock.

ppvPointer1 (OUT)

Set to a pointer to the first location in memory to write AC3 data to.

pdwBytes1 (OUT)

Set to the amount of data that can be written to *ppvPointer1*.

ppvPointer2 (OUT)

If the requested amount of memory extends beyond the end of the AC3 Data Buffer, then this pointer will point to a second piece of memory to write more AC3 data to. If this pointer is not required it will set to NULL.

pdwBytes2 (OUT)

Set to the amount of data pointed to by *ppvPointer2*.

ulFlags (IN)

Specify ENTIREBUFFER to Lock either the entire buffer (if the AC3 stream is not playing), or as much data as possible (if the AC3 stream is currently playing). Alternatively specify FROMWRITECURSOR to Lock data from the write cursor onwards.

Returns

EAXAC3_OK

EAXAC3UnLockBuffer

HRESULT EAXAC3UnLockBuffer(void **pvPointer1*, unsigned long *ulSize1*, void **pvPointer2*, unsigned long *ulSize2*, bool *bFinished*)

Description

Call this function after your application has finished writing new AC3 data to the AC3 Data buffer. Set the *ulSize1* and *ulSize2* parameters to the exact amount of data that has been written to the two locations.

If you have reached the end of the AC3 data you wish to send to the Wave device, then set the final parameter *bFinished* to *true*.

Parameters

ppvPointer1 (IN)

Pointer to first location in memory

pdwBytes1 (IN)

Set this to the amount of data that has been written to *ppvPointer1*.

ppvPointer2 (IN)

Pointer to the second location in memory (if appropriate).

pdwBytes2 (IN)

Set this to the amount of data written to *ppvPointer2*.

bFinished (IN)

Set this parameter to true if this is the end of the AC3 data.

Returns

EAXAC3_OK if successful
EAXAC3ERR_EAXAC3DEVICENOTOPEN if the EAX-AC3 Device has not been opened
EAXAC3ERR_AC3STREAMNOTOPEN if the AC3 Stream has not been opened

EAXAC3GetErrorString

char *EAXAC3GetErrorString(HRESULT *hr*, char **szErrorString*, int *nSizeOfErrorString*)

Description

Fills (and returns) the given string with a textual description of the supplied AC3 Error code for debugging purposes.

Parameters

hr (IN)

Error code

szErrorString (OUT)

String to write the textual description in

nSizeOfErrorString (IN)

Size of the *szErrorString*.

Returns

A pointer to *szErrorString*

EAXAC3GetLastError

HRESULT EAXAC3GetLastError(HRESULT **hr*)

Description

Returns the last error code generated by the EAX-AC3 API.

Parameters

hr (OUT)

Receives the last error code.

Returns

EAXAC3_OK

EAXAC3SetPlaybackMode

HRESULT EAXAC3SetPlaybackMode(EAXAC3HANDLE *EAXAC3Handle*, unsigned int *ulPlayMode*)

Description

Set the EAX-AC3 Device's playback mode. Currently defined modes are SPDIFPASSTHRU and FULLDECODE.

This function is not supported at this time.

Parameters

EAXAC3Handle (IN)

EAX-AC3 device handle.

UlPlayMode (IN)

Set to either SPDIFPASSTHRU for S/PDIF pass-thru, or FULLDECODE for the device to perform the AC3 decode itself.

Returns

EAXAC3ERR_UNSUPPORTED

Appendix 1

Code Examples

Example 1 – Playing a Dolby Digital AC3 file from disk.

```
#include "openal\alut.h"
#include "include\eaxac3.h"
#include "windows.h"
#include <stdio.h>
#include <conio.h>

// Global variables
LPEAXAC3FNTABLE g_lpEAXAC3FnTable;  // Pointer to the EAX-AC3 function table
bool g_bFinished;                   // Boolean variable to indicate when playback has completed

// Callback function to receive messages from the EAX-AC3 API
void CALLBACK AC3CallbackFn(AC3STREAM ac3Stream, int msg)
{
    switch (msg)
    {
        case EAXAC3REACHEDEND:
            printf("Received message that EOF / EOD of AC3 File has been reached !!!\n");
            g_bFinished = true;
            break;

        case EAXAC3NEEDMOREDATA:
            break;
    }
    return;
}

// Helper function to display any error messages
void DisplayError(HRESULT hr)
{
    char szErrorString[256];
    printf(g_lpEAXAC3FnTable->EAXAC3GetErrorString(hr, szErrorString, 256));
    return;
}

void CloseDown()
{
    g_lpEAXAC3FnTable->EAXAC3ClosePlaybackDevice(DEFAULTEAXAC3DEVICE);
    alutExit();
    if (g_lpEAXAC3FnTable)
        delete g_lpEAXAC3FnTable;
}

int main(int argc, char* argv[])
{
    ALboolean bEAXAC3ExtPresent;
    ALubyte szAC3[] = "EAX-AC3";
    int position, nNumberOfEAXAC3Devices = 0;
    LPALEAXAC3GETFUNCTIONTABLE ALEAXAC3GetFunctionTable;
    HRESULT hr;
    AC3STREAM ac3Stream;
    EAXAC3DEVICEINFO EAXAC3DeviceInfo;

    // Allocate memory for EAXAC3 Function table
    g_lpEAXAC3FnTable = new EAXAC3FNTABLE;

    if (g_lpEAXAC3FnTable == NULL)
    {
        printf("Out of memory\n");
        return 0;
    }

    // Initialize Open AL
    alutInit(&argc, argv);
```

```
bEAXAC3ExtPresent = alIsExtensionPresent(szAC3);

if (bEAXAC3ExtPresent)
	printf("EAX-AC3 Extension found !\n");
else
{
	printf("No EAX-AC3 Extension found\n");
	CloseDown();
	return 0;
}

// Retrieve the address of the alEAXAC3GetFunctionTable() function

ALubyte szFnName[] = "alEAXAC3GetFunctionTable";
ALEAXAC3GetFunctionTable = (LPALEAXAC3GETFUNCTIONTABLE)alGetProcAddress(szFnName);

if (ALEAXAC3GetFunctionTable == NULL)
{
	printf("Cannot find alEAXAC3GetFunctionTable function\n");
	CloseDown();
	return 0;
}

// Use the ALEAXAC3GetFunctionTable() function to retrieve a function table containing all of the
// EAX-AC3 Functions
if (!ALEAXAC3GetFunctionTable(g_lpEAXAC3FnTable))
{
	printf("Failed to retrieve the EAX-AC3 Function Table\n");
	CloseDown();
	return 0;
}

// Find out how many AC3 Devices are available
nNumberOfEAXAC3Devices = g_lpEAXAC3FnTable->EAXAC3QueryNumberOfDevices();

printf("Number of EAX-AC3 Playback devices is %d\n", nNumberOfEAXAC3Devices);

// Get the capabilities of each EAX-AC3 Playback device
for (int loop = 0; loop < nNumberOfEAXAC3Devices; loop++)
{
	if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3QueryDeviceCaps(loop, &EAXAC3DeviceInfo,
		sizeof(EAXAC3DEVICEINFO))))
	{
		DisplayError(hr);
		CloseDown();
		return 0;
	}

	printf("EAX-AC3 Device name is %s\n", EAXAC3DeviceInfo.szDeviceName);

	printf("Number Of AC3 Streams supported = %d\n", EAXAC3DeviceInfo.uStreams);

	if ((EAXAC3DeviceInfo.uFlags & SPDIFPASSTHRU) == SPDIFPASSTHRU)
		printf("Device supports AC3 SP/DIF Pass thru\n");
	if ((EAXAC3DeviceInfo.uFlags & FULLDECODE) == FULLDECODE)
		printf("Device supports AC3 Decode\n");
	if ((EAXAC3DeviceInfo.uFlags & UNKNOWN) == UNKNOWN)
		printf("Cannot tell if the device supports SP/DIF Pass-thru and / or AC3 Decode\n");
}

// Open the default AC3 Playback Device
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3OpenPlaybackDevice(DEFAULTEAXAC3DEVICE)))
{
	DisplayError(hr);
	CloseDown();
	return 0;
}

// Find out the specifications of the AC3 File
AC3FILEINFO ac3FileInfo;
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3QueryFile("testtechno.ac3", &ac3FileInfo, sizeof(AC3FILEINFO))))
{
	DisplayError(hr);
	CloseDown();
}

printf("AC3Frame Size %d\nNo. Of AC3Frames %d\nFrequency %d\n", ac3FileInfo.nAC3FrameSize,
            ac3FileInfo.nNumOfAC3Frames, ac3FileInfo.nFrequency);
printf("Duration %d Seconds\nSize Of file %d bytes\n", ac3FileInfo.nDuration / 1000, ac3FileInfo.nSizeOfFile);

// Open an AC3FILE (testtechno.ac3) AC3 Stream, specifying a callback function to receive a message when
// playback has completed
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3OpenStream(DEFAULTEAXAC3DEVICE, &ac3Stream,
       &AC3CallbackFn, "testtechno.ac3", AC3FILE)))
{
      DisplayError(hr);
      CloseDown();
      return 0;
}

int nAC3FramesToSkip;
char c;

// Set position in AC3 File
printf("Enter number of AC3 Frames to skip : ");
scanf("%d", &nAC3FramesToSkip);

if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3SetFilePosition(ac3Stream, AC3FRAMES, nAC3FramesToSkip)))
{
      DisplayError(hr);
      CloseDown();
      return false;
}

printf("\nSkipped %d AC3 Frames, press a key to start playback\n", nAC3FramesToSkip);
c = getch();

// Play AC3 Stream once
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3PlayStream(ac3Stream, false)))
{
      DisplayError(hr);
      CloseDown();
      return 0;
}

// Continuously call EAXAC3GetPosition until we receive a message that the file has finished playing,
// or the user presses a key

printf("Press a key to Stop Playback\n");
while (!_kbhit() && !g_bFinished)
{
      g_lpEAXAC3FnTable->EAXAC3GetPosition(ac3Stream, AC3FRAMES, &position);
      printf("Position in AC3 Frames from start position is %d\r", position);
}

// Stop AC3 Stream playing
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3StopStream(ac3Stream)))
{
      DisplayError(hr);
      CloseDown();
      return 0;
}

// Close AC3 Stream
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3CloseStream(ac3Stream)))
{
      DisplayError(hr);
      CloseDown();
      return 0;
}

// Close EAX-AC3 Playback device
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3ClosePlaybackDevice(DEFAULTEAXAC3DEVICE)))
{
      DisplayError(hr);
      CloseDown();
      return 0;
}
```

```
        // Close down Open AL
        alutExit();

        // Release memory allocated for EAX-AC3 Function table
        delete g_lpEAXAC3FnTable;

        return 0;
}
```

Example 2 – Streaming a Dolby Digital AC3 file from memory – using ‘Push’ mode.

```
#include "alut.h"
#include "eaxac3.h"
#include "windows.h"
#include <stdio.h>
#include <conio.h>

// Global variables
LPEAXAC3FNTABLE g_lpEAXAC3FnTable;  // Pointer to the EAX-AC3 function table
HANDLE g_hEvent;                    // Event handle
FILE *AC3File;                      // AC3 File Pointer

// Callback function to receive messages from the EAX-AC3 API
void CALLBACK AC3CallbackFn(AC3STREAM ac3Stream, int msg)
{
    switch (msg)
    {
        case EAXAC3REACHEDEND:
            printf("Received message that EOF / EOD of AC3 File has been reached !!!\n");
            SetEvent(g_hEvent);
            break;

        case EAXAC3NEEDMOREDATA:
            break;
    }
    return;
}

// Helper function to display any error messages
void DisplayError(HRESULT hr)
{
    char szErrorString[256];
    printf(g_lpEAXAC3FnTable->EAXAC3GetErrorString(hr, szErrorString, 256));
    return;
}

void CloseDown()
{
    g_lpEAXAC3FnTable->EAXAC3ClosePlaybackDevice(DEFAULTEAXAC3DEVICE);
    alutExit();
    if (AC3File != NULL)
        fclose(AC3File);
    if (g_lpEAXAC3FnTable)
        delete g_lpEAXAC3FnTable;
    if (g_hEvent)
    {
        CloseHandle(g_hEvent);
        g_hEvent = 0;
    }
}

int main(int argc, char* argv[])
{
    ALboolean bEAXAC3ExtPresent;
    ALubyte szAC3[] = "EAX-AC3";
    int nNumberOfEAXAC3Devices = 0;
    LPALEAXAC3GETFUNCTIONTABLE ALEAXAC3GetFunctionTable;
    HRESULT hr;
    AC3STREAM ac3Stream;
    unsigned long nAC3DataSize;
    unsigned long nAC3DataRead = 0;
    DWORD dwSize1, dwSize2, dwBytesRead;
    LPVOID pBuffer1, pBuffer2;
    AC3FILEINFO   ac3FileInfo;
    char c;
```

```
// Initialize Global variables
g_hEvent = NULL;
g_lpEAXAC3FnTable = NULL;

// Create an Event to be set when playback has completed
SECURITY_ATTRIBUTES     sattrib;
sattrib.nLength = sizeof(SECURITY_ATTRIBUTES);
sattrib.bInheritHandle = false;
sattrib.lpSecurityDescriptor = NULL;

// Create a Win32 event to notify our application when playback has completed
g_hEvent = CreateEvent(&sattrib, true, false, "FinishedPlayback");

if (g_hEvent == NULL)
{
      printf("Failed to create Win32 Event\n");
      return 0;
}

// Allocate memory for EAX-AC3 function table
g_lpEAXAC3FnTable = new EAXAC3FNTABLE;

if (g_lpEAXAC3FnTable == NULL)
{
      printf("Out of memory\n");
      return 0;
}

// Initialize Open AL
alutInit(&argc, argv);

bEAXAC3ExtPresent = alIsExtensionPresent(szAC3);

if (bEAXAC3ExtPresent)
      printf("EAX-AC3 Extension found !\n");
else
{
      printf("No EAX-AC3 Extension found\n");
      CloseDown();
      return 0;
}

// Retrieve the address of the alEAXAC3GetFunctionTable() function

ALubyte szFnName[] = "alEAXAC3GetFunctionTable";
ALEAXAC3GetFunctionTable = (LPALEAXAC3GETFUNCTIONTABLE)alGetProcAddress(szFnName);

if (ALEAXAC3GetFunctionTable == NULL)
{
      printf("Cannot find alEAXAC3GetFunctionTable function\n");
      CloseDown();
      return 0;
}

// Use the ALEAXAC3GetFunctionTable() function to retrieve a function table containing all of the
// EAX-AC3 Functions
if (!ALEAXAC3GetFunctionTable(g_lpEAXAC3FnTable))
{
      printf("Failed to retrieve the EAX-AC3 Function Table\n");
      CloseDown();
      return 0;
}

// Open the Default EAX-AC3 Device
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3OpenPlaybackDevice(DEFAULTEAXAC3DEVICE)))
{
      DisplayError(hr);
      CloseDown();
      return 0;
}

// Open AC3 stream from memory - with callback fn, and using ‘Push’ mode
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3OpenStream(0, &ac3Stream, &AC3CallbackFn, "PUSH", MEMORY)))
```

```
        DisplayError(hr);
        CloseDown();
        return false;
}

// Query for information about the AC3 File
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3QueryFile("dolby.ac3", &ac3FileInfo, sizeof(AC3FILEINFO))))
{
        DisplayError(hr);
        CloseDown();
        return false;
}

// Display CAPS
printf("Number Of AC3 Frames is %d\nAC3Frame Size = %dbytes\n", ac3FileInfo.nNumOfAC3Frames,
        ac3FileInfo.nAC3FrameSize);
printf("Duration of AC3 File = %dSecs\nFrequency = %dHz\nFile Size = %d\n", ac3FileInfo.nDuration / 1000,
        ac3FileInfo.nFrequency, ac3FileInfo.nSizeOfFile);

nAC3DataSize = ac3FileInfo.nSizeOfFile;

AC3File = fopen("dolby.ac3", "rb");

if (AC3File == NULL)
{
        printf("Cannot find .ac3 file\n");
        CloseDown();
        return false;
}

// Start by filling AC3 buffer with data
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3LockBuffer(ac3Stream, 0, &pBuffer1, &dwSize1, &pBuffer2,
        &dwSize2, ENTIREBUFFER)))
{
        DisplayError(hr);
        CloseDown();
        return false;
}

fread(pBuffer1, sizeof(char), dwSize1, AC3File);

nAC3DataRead += dwSize1;
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3UnLockBuffer(ac3Stream, pBuffer1, dwSize1, pBuffer2, dwSize2,
        false)))
{
        DisplayError(hr);
        CloseDown();
        return false;
}

printf("Press a key to start playback\n");
c = getch();

if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3PlayStream(ac3Stream, 0)))
{
        if (hr == EAXAC3ERR_NOTENOUGHAC3DATAINAC3DATABUFFER)
                printf("AC3 Frames required is %d, Equivalent to %dbytes\n",
                  g_lpEAXAC3FnTable->EAXAC3QueryNoOfFramesReqForPlayback (ac3Stream),
                ac3FileInfo.nAC3FrameSize*g_lpEAXAC3FnTable->EAXAC3QueryNoOfFramesReqForPlayback(ac3Stream));
        DisplayError(hr);
        CloseDown();
        return false;
}

printf("Press a key to Stop\n");

// Try and write 10000 bytes of AC3 data each iteration of the following loop
while ( (!bEOF) && !_kbhit() )
{
        if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3LockBuffer(ac3Stream, 0, &pBuffer1, &dwSize1, &pBuffer2,
                &dwSize2, ENTIREBUFFER))) // FROMWRITECURSOR)))
        {
                DisplayError(hr);
                CloseDown();
```

```
		}

		// If there is room for more AC3 data, then either dwSize1 or dwSize2 will be greater than 0
		if ((dwSize1 != 0) || (dwSize2 != 0))
		{
			// Room for more AC3 data

			// Check that we have enough data to read from the file.  If there isn't enough, then read
			// what is left, and set our EOF flag to true
			if ((nAC3DataRead + dwSize1) > nAC3DataSize)
			{
				dwSize1 = nAC3DataSize - nAC3DataRead;
				bEOF = true;
			}

			dwBytesRead = fread(pBuffer1, sizeof(char), dwSize1, AC3File);
			if (dwBytesRead != dwSize1)
			{
				printf("Error reading from file\n");
				CloseDown();
				return false;
			}

			nAC3DataRead += dwSize1;

			// If we haven't reached the EOF and dwSize is greater than 0
			if ((!bEOF) && (dwSize2 != 0))
			{
				// Check if there is enough data to fill the second part of the Locked memory
				if ((nAC3DataRead + dwSize2) > nAC3DataSize)
				{
					// There isn't enough data - just read what data is left, and set our EOF flag to true
					dwSize2 = nAC3DataSize - nAC3DataRead;
					bEOF = true;
				}

				dwBytesRead = fread(pBuffer2, sizeof(char), dwSize2, AC3File);
				if (dwBytesRead != dwSize2)
				{
					printf("Error reading from file\n");
					CloseDown();
					return false;
				}

				nAC3DataRead += dwSize2;
			}
		}

		// UnLock the data supplying values for the amount of data we actually wrote
		// The last parameter indicates whether the end of the AC3 data has been reached
		if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3UnLockBuffer(ac3Stream, pBuffer1, dwSize1, pBuffer2, dwSize2,
			bEOF)))
		{
			DisplayError(hr);
			CloseDown();
			return false;
		}
	}	// End of while loop

	if (!bEOF)
		c = getch();	// swallow input
	else
	{
		// Reached end of .ac3 file - wait until all the data has been played, before stopping the stream
		printf("Reached end of .ac3 file.  Waiting for notification that playback has ended\n");

		// Wait for the event to be signalled that playback has completed (this is Set by our callback fn)
		WaitForSingleObjectEx(g_hEvent, 5000, false);		// Time out at 5 Seconds - just in case ...

		printf("\nFinished playback !\n");
	}

	if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3StopStream(ac3Stream)))
	{
			CloseDown();
			return false;
		}

		// Close AC3 Stream
		if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3CloseStream(ac3Stream)))
		{
			DisplayError(hr);
			CloseDown();
			return false;
		}

		// Close AC3 Device
		if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3ClosePlaybackDevice(DEFAULTEAXAC3DEVICE)))
		{
			DisplayError(hr);
			CloseDown();
			return 0;
		}

		// Close AC3 File
		fclose(AC3File);

		// Close Open AL
		alutExit();

		// Delete memory allocated for EAX-AC3 Function table
		delete g_lpEAXAC3FnTable;

		// Destroy the Event
		if (g_hEvent)
			CloseHandle(g_hEvent);

		return 0;
	}
```

Example 3 – Streaming a Dolby Digital AC3 file from memory – using ‘Pull’ mode.

```
#include "openal\alut.h"
#include "include\eaxac3.h"
#include "windows.h"
#include <stdio.h>
#include <conio.h>

// Global variables
LPEAXAC3FNTABLE g_lpEAXAC3FnTable;  // Pointer to the EAX-AC3 function table
bool g_bFinished;                   // Boolean variable to indicate when playback has completed
FILE * g_AC3File;                   // File pointer
unsigned int g_nAC3DataRead;        // Data read from file
unsigned int g_nAC3DataSize;        // Size of file

// Helper function to display any error messages
void DisplayError(HRESULT hr)
{
    char szErrorString[256];
    printf(g_lpEAXAC3FnTable->EAXAC3GetErrorString(hr, szErrorString, 256));
    return;
}

void CloseDown()
{
    g_lpEAXAC3FnTable->EAXAC3ClosePlaybackDevice(DEFAULTEAXAC3DEVICE);
    alutExit();
    if (g_AC3File != NULL)
        fclose(g_AC3File);
    if (g_lpEAXAC3FnTable)
        delete g_lpEAXAC3FnTable;
}

// Callback function to receive messages from the EAX-AC3 API
void CALLBACK AC3CallbackFn(AC3STREAM ac3Stream, int msg)
{
    HRESULT hr;
    DWORD dwSize1, dwSize2, dwBytesRead;
    LPVOID pBuffer1, pBuffer2;
    bool bEOF = false;

    switch (msg)
    {
        case EAXAC3REACHEDEND:
            printf("Received message that EOF / EOD of AC3 File has been reached !!!\n");
            g_bFinished = true;
            break;

        case EAXAC3NEEDMOREDATA:
            // Top the buffer up with more AC3 Data
            if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3LockBuffer(ac3Stream, 0, &pBuffer1, &dwSize1, &pBuffer2,
                &dwSize2, ENTIREBUFFER)))
            {
                DisplayError(hr);
                CloseDown();
                return;
            }

            // If there is room for more AC3 data, then either dwSize1 or dwSize2 will be greater than 0
            if ((dwSize1 != 0) | (dwSize2 != 0))
            {
                // Check that we have enough data to read from the file.  If there isn't enough, then read
                // what is left, and set our EOF flag to true
                if ((g_nAC3DataRead + dwSize1) > g_nAC3DataSize)
                {
                    dwSize1 = g_nAC3DataSize - g_nAC3DataRead;
                }

                dwBytesRead = fread(pBuffer1, sizeof(char), dwSize1, g_AC3File);
                if (dwBytesRead != dwSize1)
                {
                    printf("Error reading from file\n");
                    CloseDown();
                    return;
                }

                g_nAC3DataRead += dwSize1;

                // If we haven't reached the EOF and dwSize is greater than 0
                if ((!bEOF) & (dwSize2 != 0))
                {
                    // Check if there is enough data to fill the second part of the Locked memory
                    if ((g_nAC3DataRead + dwSize2) > g_nAC3DataSize)
                    {
                        // There isn't enough data - just read what data is left, and set our EOF flag to true
                        dwSize2 = g_nAC3DataSize - g_nAC3DataRead;
                        bEOF = true;
                    }

                    dwBytesRead = fread(pBuffer2, sizeof(char), dwSize2, g_AC3File);
                    if (dwBytesRead != dwSize2)
                    {
                        printf("Error reading from file\n");
                        CloseDown();
                        return;
                    }

                    g_nAC3DataRead += dwSize2;
                }
            }

            // UnLock the data supplying values for the amount of data we actually wrote
            // The last parameter indicates whether the end of the AC3 data has been reached
            g_lpEAXAC3FnTable->EAXAC3UnLockBuffer(ac3Stream, pBuffer1, dwSize1, pBuffer2, dwSize2, bEOF);

            break;

        default:
            break;
    }

    return;
}


int main(int argc, char* argv[])
{
    ALboolean bEAXAC3ExtPresent;
    ALubyte szAC3[] = "EAX-AC3";
    int nNumberOfEAXAC3Devices = 0;
    LPALEAXAC3GETFUNCTIONTABLE ALEAXAC3GetFunctionTable;
    HRESULT hr;
    AC3STREAM ac3Stream;
    int position;
    DWORD dwSize1, dwSize2, dwBytesRead;
    LPVOID pBuffer1, pBuffer2;
    AC3FILEINFO   ac3FileInfo;
    char c;
    bool bEOF = false;

    // Initialize global variables
    g_AC3File = NULL;
    g_bFinished = false;
    g_lpEAXAC3FnTable = NULL;

    // Allocate memory for EAX-AC3 function table
    g_lpEAXAC3FnTable = new EAXAC3FNTABLE;

    if (g_lpEAXAC3FnTable == NULL)
    {
```

```
    return 0;
}

// Initialize Open AL
alutInit(&argc, argv);

bEAXAC3ExtPresent = alIsExtensionPresent(szAC3);

if (bEAXAC3ExtPresent)
    printf("EAX-AC3 Extension found !\n");
else
{
    printf("No EAX-AC3 Extension found\n");
    CloseDown();
    return 0;
}

// Retrieve the address of the alEAXAC3GetFunctionTable() function

ALubyte szFnName[] = "alEAXAC3GetFunctionTable";
ALEAXAC3GetFunctionTable = (LPALEAXAC3GETFUNCTIONTABLE)alGetProcAddress(szFnName);

if (ALEAXAC3GetFunctionTable == NULL)
{
    printf("Cannot find alEAXAC3GetFunctionTable function\n");
    CloseDown();
    return 0;
}

// Use the ALEAXAC3GetFunctionTable() function to retrieve a function table containing all of the
// EAX-AC3 Functions
if (!ALEAXAC3GetFunctionTable(g_lpEAXAC3FnTable))
{
    printf("Failed to retrieve the EAX-AC3 Function Table\n");
    CloseDown();
    return 0;
}

// Open the Default EAX-AC3 Device
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3OpenPlaybackDevice(DEFAULTEAXAC3DEVICE)))
{
    DisplayError(hr);
    CloseDown();
    return 0;
}

// Open AC3 stream from memory - with callback fn, using ‘Pull’ mode
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3OpenStream(0, &ac3Stream, &AC3CallbackFn, "PULL", MEMORY)))
{
    DisplayError(hr);
    CloseDown();
    return false;
}

// Get CAPS of AC3 File
if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3QueryFile("kirdee.ac3", &ac3FileInfo, sizeof(AC3FILEINFO))))
{
    DisplayError(hr);
    CloseDown();
    return false;
}

// Display CAPS
printf("Number Of AC3 Frames is %d\nAC3Frame Size = %dbytes\n", ac3FileInfo.nNumOfAC3Frames,
    ac3FileInfo.nAC3FrameSize);
printf("Duration of AC3 File = %dSecs\nFrequency = %dHz\nFile Size = %d\n", ac3FileInfo.nDuration / 1000,
    ac3FileInfo.nFrequency, ac3FileInfo.nSizeOfFile);

g_nAC3DataSize = ac3FileInfo.nSizeOfFile;

g_AC3File = fopen("kirdee.ac3", "rb");

if (g_AC3File == NULL)
{
        CloseDown();
        return false;
    }

    // Start by filling AC3 buffer with data
    if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3LockBuffer(ac3Stream, 0, &pBuffer1, &dwSize1, &pBuffer2,
        &dwSize2, ENTIREBUFFER)))
    {
        DisplayError(hr);
        CloseDown();
        return false;
    }

    dwBytesRead = fread(pBuffer1, sizeof(char), dwSize1, g_AC3File);

    g_nAC3DataRead += dwSize1;

    g_lpEAXAC3FnTable->EAXAC3UnLockBuffer(ac3Stream, pBuffer1, dwSize1, pBuffer2, dwSize2, false);

    // Test QueryAC3Memory()
    if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3QueryMemory((char*)pBuffer1, dwSize1, &ac3FileInfo,
        sizeof(AC3FILEINFO))))
    {
        DisplayError(hr);
        CloseDown();
        return false;
    }

    // Display CAPS
    printf("Number Of AC3 Frames is %d\nAC3Frame Size = %dbytes\n", ac3FileInfo.nNumOfAC3Frames,
        ac3FileInfo.nAC3FrameSize);
    printf("Duration of AC3 File = %dms\nFrequency = %dHz\nFile Size = %d\n", ac3FileInfo.nDuration,
        ac3FileInfo.nFrequency, ac3FileInfo.nSizeOfFile);

    printf("Press a key to start playback\n");
    c = getch();

    if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3PlayStream(ac3Stream, 0)))
    {
        if (hr == EAXAC3ERR_NOTENOUGHAC3DATAINAC3DATABUFFER)
            printf("AC3 Frames required is %d, Equivalent to %dbytes\n",
             g_lpEAXAC3FnTable->EAXAC3QueryNoOfFramesReqForPlayback(ac3Stream),
            ac3FileInfo.nAC3FrameSize*g_lpEAXAC3FnTable->EAXAC3QueryNoOfFramesReqForPlayback(ac3Stream));

        DisplayError(hr);
        CloseDown();
        return false;
    }

    printf("Press a key to Stop\n");
    while ( (!g_bFinished) && !_kbhit() )
    {
        g_lpEAXAC3FnTable->EAXAC3GetPosition(ac3Stream, MILLISECONDS, &position);
        printf("Position is %d Seconds\r", position / 1000);
    }

    if (!g_bFinished)
        c = getch();   // swallow input

    if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3StopStream(ac3Stream)))
    {
        DisplayError(hr);
        CloseDown();
        return false;
    }
    // Close AC3 Stream
    if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3CloseStream(ac3Stream)))
    {
        DisplayError(hr);
        CloseDown();
        return false;
    }
    // Close EAX-AC3 device
    if (FAILED(hr = g_lpEAXAC3FnTable->EAXAC3ClosePlaybackDevice(DEFAULTEAXAC3DEVICE)))
```

```
            DisplayError(hr);
            CloseDown();
            return false;
        }
        // Close AC3 File
        fclose(g_AC3File);
        // Shutdown Open AL
        alutExit();
        // Release memory allocated for EAX-AC3 function table
        delete g_lpEAXAC3FnTable;
        return true;
}
```