DirectSound

[This is preliminary documentation and subject to change.]

This section provides information about the DirectSound® component of the Microsoft DirectX® application programming interface (API). Information is found under the following main headings:

- About DirectSound
- Why Use DirectSound?
- DirectSound Architecture
- DirectSound Essentials
- DirectSound Tutorials
- DirectSound Reference
- DirectSound Samples

About DirectSound

[This is preliminary documentation and subject to change.]

The Microsoft® DirectSound® application programming interface (API) is the wave-audio component of the DirectX® API. DirectSound provides low-latency mixing, hardware acceleration, and direct access to the sound device. It provides this functionality while maintaining compatibility with existing device drivers.

DirectSound enables wave sound capture and playback. It also supports property sets, which enable application developers to take advantage of extended services offered by sound cards and their associated drivers.

Why Use DirectSound?

[This is preliminary documentation and subject to change.]

The overriding design goal in DirectX is speed. Like other components of DirectX, DirectSound allows you to use the hardware in the most efficient way possible while insulating you from the specific details of that hardware with a device-independent interface. Your applications will work well with the simplest audio hardware but will also take advantage of the special features of cards and drivers that have been enhanced for use with DirectSound.

Here are some other things that DirectSound makes easy:

- Querying hardware capabilities at run time to determine the best solution for any given personal computer configuration

- Using property sets so that new hardware capabilities can be exploited even when they are not directly supported by DirectSound
- Low-latency mixing of audio streams for rapid response
- Implementing three dimensional (3-D) sound
- Capturing sound

Despite the advantages of DirectSound, the standard waveform-audio functions in Windows® continue to be a practical solution for certain tasks. For example, an application can easily play a single sound or *audio stream*, such as introductory music, by using the **PlaySound** or **waveOut** functions.

DirectSound Architecture

[This is preliminary documentation and subject to change.]

This section introduces the components of DirectSound and explains how DirectSound works with the hardware and with other applications. The following topics are discussed:

- Architectural Overview
- Sound Data
- Playback Overview
- Capture Overview
- Property Sets Overview
- Hardware Abstraction and Emulation
- System Integration

Architectural Overview

[This is preliminary documentation and subject to change.]

DirectSound implements a new model for playing and capturing digital sound samples and mixing sample sources. Like other elements of the DirectX API, DirectSound uses the hardware to its greatest advantage whenever possible, and it emulates hardware features in software when the feature is not present in the hardware.

DirectSound playback is built on the **IDirectSound** Component Object Model (COM) interface and on other interfaces for manipulating sound buffers and 3-D effects. These interfaces are **IDirectSoundBuffer**, **IDirectSound3DBuffer**, and **IDirectSound3DListener**

DirectSound capture is based on the **IDirectSoundCapture** and **IDirectSoundCaptureBuffer** COM interfaces.

Another COM interface, **IKsPropertySet**, provides methods that allow applications to take advantage of extended capabilities of sound cards, even those introduced in the future.

Finally, the **IDirectSoundNotify** interface is used to signal events when playback or capture has reached a certain point in a buffer.

For more information about COM concepts that you should understand to create applications with the DirectX APIs, see The Component Object Model.

Sound Data

[This is preliminary documentation and subject to change.]

DirectSound and DirectSoundCapture work with waveform audio data, which consists of digital samples of the sound at a fixed frequency. The particular format of a sound can be described by a **WAVEFORMATEX** structure. This structure is documented in the Multimedia Structures section of the Platform SDK documentation, but is briefly described here for convenience:

```
typedef struct {
    WORD  wFormatTag;
    WORD  nChannels;
    DWORD nSamplesPerSec;
    DWORD nAvgBytesPerSec;
    WORD  nBlockAlign;
    WORD  wBitsPerSample;
    WORD  cbSize;
} WAVEFORMATEX;
```

The **wFormatTag** member contains a unique identifier assigned by Microsoft Corporation. A complete list can be found in the Mmreg.h header file. The only tag valid with DirectSound is WAVE_FORMAT_PCM. This tag indicates Pulse Code Modulation (PCM), an uncompressed format in which each samples represents the amplitude of the signal at the time of sampling. DirectSoundCapture can capture data in other formats by using the Audio Compression Manager.

For information on using non-PCM data with DirectSound, see Compressed Wave Formats.

The **nChannels** member describes the number of channels, usually either one (mono) or two (stereo). For stereo data, the samples are interleaved. The **nSamplesPerSec** member describes the sampling rate, or frequency, in hertz. Typical values are 11,025, 22,050, and 44,100.

The **wBitsPerSample** member gives the size of each sample, generally 8 or 16 bits. The value in **nBlockAlign** is the number of bytes required for each complete sample, and for PCM formats is equal to (**wBitsPerSample** * **nChannels** / 8). The value in **nAvgBytesPerSec** is the product of **nBlockAlign** and **nSamplesPerSec**.

Finally, **cbSize** gives the size of any extra fields required to describe a specialized wave format. This member is always zero for PCM formats.

Playback Overview

[This is preliminary documentation and subject to change.]

The DirectSound buffer object represents a buffer containing sound data. Buffer objects are used to start, stop, and pause sound playback, as well as to set attributes such as frequency and format.

The *primary sound buffer* holds the audio that the listener will hear. *Secondary sound buffers* each contain a single sound or stream of audio. DirectSound automatically creates a primary buffer, but it is the application's responsibility to create secondary buffers. When sounds in secondary buffers are played, DirectSound mixes them in the primary buffer and sends them to the output device. Only the available processing time limits the number of buffers that DirectSound can mix.

It is your responsibility to stream data in the correct format into the secondary sound buffers. DirectSound does not include methods for parsing a sound file or a wave resource. However, there is code in the accompanying sample applications that will help you with this task. For more information, see Using Wave Files and Reading Wave Data from a Resource.

Depending on the card type, DirectSound buffers can exist in hardware as on-board RAM, wave-table memory, a direct memory access (DMA) channel, or a virtual buffer (for an I/O port based audio card). Where there is no hardware implementation of a DirectSound buffer, it is emulated in system memory.

Multiple applications can create DirectSound objects for the same sound device. When the input focus changes between applications, the audio output automatically switches from one application's streams to another's. As a result, applications do not have to repeatedly play and stop their buffers when the input focus changes.

In order to know when a streaming buffer is ready to receive new data, or when any buffer has stopped, an application can use the **IDirectSoundNotify** interface to set up notification positions. When the play cursor reaches one of these positions, an event is signaled. Alternatively, an application can regularly poll the position of the play cursor.

Capture Overview

[This is preliminary documentation and subject to change.]

The DirectSoundCapture object is used to query the capabilities of sound capture devices and to create buffers for capturing audio from an input source. DirectSoundCapture allows capturing of data in PCM or compressed formats.

The DirectSoundCaptureBuffer object represents a buffer used for receiving data from the input device. Like playback buffers, this buffer is conceptually circular:

when input reaches the end of the buffer, it automatically starts again at the beginning.

The methods of the DirectSoundCaptureBuffer object allow you to retrieve the properties of the buffer, start and stop audio capture, and lock portions of the memory so that you can safely retrieve data for saving to a file or for some other purpose.

As with playback, DirectSound allows you to request notification when captured data reaches a specified position within the buffer, or when capture has stopped. This service is provided through the **IDirectSoundNotify** interface.

Property Sets Overview

[This is preliminary documentation and subject to change.]

Through property sets, DirectSound is able to support extended services offered by manufacturers of sound cards and their associated drivers.

Hardware vendors define new capabilities as properties and publish the specification for these properties. You, the application developer, can then use the methods of the **IKsPropertySet** interface to determine whether a particular set of properties is available on the target hardware and to manipulate those properties, for instance by turning special effects on and off.

Property sets allow for the unlimited extension of the capabilities of DirectSound. You use a single method, **IKsPropertySet::Set**, to alter the state of the device in any way specified by the manufacturer.

Hardware Abstraction and Emulation

[This is preliminary documentation and subject to change.]

DirectSound accesses the sound hardware through the DirectSound hardware abstraction layer (HAL), an interface that is implemented by the audio-device driver.

The DirectSound HAL provides the following functionality:

- Acquires and releases control of the audio hardware
- Describes the capabilities of the audio hardware
- Performs the specified operation when hardware is available
- Causes the operation request to report failure when hardware is unavailable

The device driver does not perform any software emulation; it simply reports the capabilities of the hardware to DirectSound and passes requests from DirectSound to the hardware. If the hardware cannot perform a requested operation, the device driver reports failure of the request and DirectSound emulates the operation.

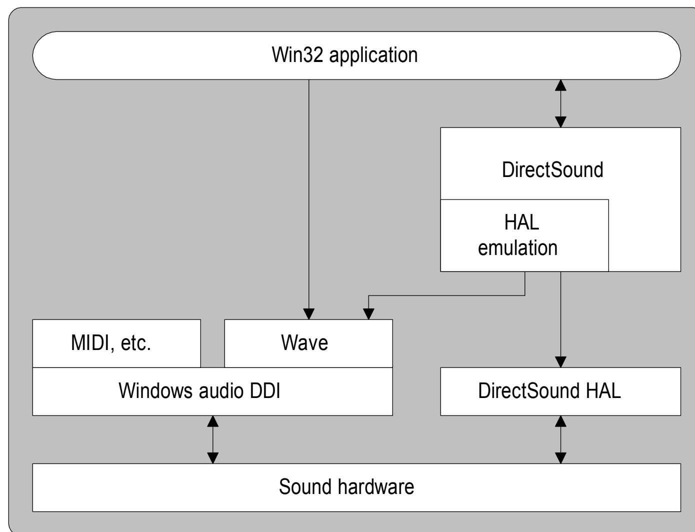
Your application can use DirectSound as long as the DirectX run-time files are present on the user's system. If the sound hardware does not have an installed DirectSound driver, DirectSound uses its hardware emulation layer (HEL), which employs the Windows multimedia waveform-audio (**waveIn** and **waveOut**) functions. Most DirectSound features are still available through the HEL, but of course hardware acceleration is not possible.

DirectSound automatically takes advantage of accelerated sound hardware, including hardware mixing and hardware sound-buffer memory. Your application need not query the hardware or program specifically to use hardware acceleration. However, for you to make the best possible use of the available hardware resources, you can query DirectSound at run time to receive a full description of the capabilities of the sound device, and then use different routines optimized for the presence or absence of a given feature. You can also specify which sound buffers should receive hardware acceleration.

System Integration

[This is preliminary documentation and subject to change.]

The following illustration shows the relationships between DirectSound and other system audio components.



DirectSound and the standard Windows waveform-audio functions provide alternative paths to the waveform-audio portion of the sound hardware. A single device provides access from one path at a time. If a waveform-audio driver has allocated a device, an attempt to allocate that same device by using DirectSound will fail. Similarly, if a DirectSound driver has allocated a device, an attempt to allocate the device by using the waveform-audio driver will fail.

If two sound devices are installed in the system, your application can access each device independently through either DirectSound or the waveform-audio functions.

Note

Microsoft Video for Windows currently uses the waveform-audio functions to play the audio track of an audio visual interleaved (.avi) file. Therefore, if your application is using DirectSound and you play an .avi file, the audio track will not be audible. Similarly, if you play an .avi file and attempt to create a DirectSound object, the creation function will return an error.
For now, applications can release the DirectSound object by calling **IDirectSound::Release** before playing an .avi file. Applications can then re-create and reinitialize the DirectSound object and its DirectSoundBuffer objects when the video finishes playing.

DirectSound Essentials

[This is preliminary documentation and subject to change.]

This section gives a practical overview of how the various DirectSound interfaces are used in order to play and capture sound. The following topics are discussed:

- DirectSound Devices
- DirectSound Buffers
- DirectSound in 3-D
- DirectSound 3-D Buffers
- DirectSound 3-D Listeners
- DirectSoundCapture
- DirectSound Property Sets
- Optimizing DirectSound Performance
- Using Wave Files
- Reading Wave Data from a Resource

Note

Most of the examples of method calls throughout this section are given in the C language form, which accesses methods by means of a pointer to a table of pointers to functions, and requires a *this* pointer as the first parameter in any call. For example, a C call to the **IDirectSound::GetCaps** method takes this form:

```
lpDirectSound->lpVtbl->GetCaps(lpDirectSound, &dscaps);
```

The same method call in the C++ form, which treats COM interface methods just like class methods, looks like this:

```
lpDirectSound->GetCaps(&dscaps);
```

Dsound.h contains macros that expand to either the C or C++ form of the method call, depending on the environment. These macros simplify the C calls and also make it possible to develop routines that can be used in either language. For example:

```
IDirectSound_GetCaps(lpDirectSound, &dscaps);
```

DirectSound Devices

[This is preliminary documentation and subject to change.]

The first step in implementing DirectSound in an application is to create a DirectSound object, which represents a sound device and gives access to the **IDirectSound** interface.

This section describes how your application can enumerate available sound devices, create the DirectSound object for a device, and use the methods of the object to set the cooperative level, retrieve the capabilities of the device, create sound buffers, set the configuration of the system's speakers, and compact hardware memory.

- Enumeration of Sound Devices
- Creating the DirectSound Object
- Cooperative Levels
- Device Capabilities
- Speaker Configuration
- Compacting Hardware Memory

Enumeration of Sound Devices

[This is preliminary documentation and subject to change.]

For an application that is simply going to play sounds through the user's preferred playback device, you don't need to enumerate the available devices. When you create the DirectSound object with NULL as the device identifier, the interface will automatically be associated with the default device if one is present. (If no device driver is present, the call to the **DirectSoundCreate** function fails.)

However, if you are looking for a particular kind of device, wish to offer the user a choice of devices, or need to work with two or more devices, you must get DirectSound to enumerate the devices available on the system.

Enumeration serves three purposes:

- Reports what hardware is available
- Supplies a globally unique identifier (GUID) for each device

- Allows you to initialize DirectSound for each device as it is enumerated, so that you can check the capabilities of the device

To enumerate devices you must first set up a callback function that will be called each time DirectSound finds a device. You can do anything you want within this function, and you can give it any name, but you must declare it in the same form as **DSEnumCallback**, a prototype in this documentation. The callback function must return TRUE if enumeration is to continue, or FALSE otherwise (for instance, after finding a device with the capabilities you need).

The following callback function, extracted from Dsenum.c in the Dsshow sample, adds information about each enumerated device to a combo box. Note that the first three parameters are supplied by the device driver. The fourth parameter is passed on from the **DirectSoundEnumerate** function.

```
BOOL CALLBACK DSEnumProc(LPGUID lpGUID,
                 LPCTSTR lpszDesc,
                 LPCTSTR lpszDrvName,
                 LPVOID lpContext )
{
    HWND   hCombo = *(HWND *)lpContext;
    LPGUID lpTemp = NULL;

    if ( lpGUID != NULL )
    {
        if (( lpTemp = LocalAlloc( LPTR, sizeof(GUID))) == NULL )
        return( TRUE );

        memcpy( lpTemp, lpGUID, sizeof(GUID));
    }

    ComboBox_AddString( hCombo, lpszDesc );
    ComboBox_SetItemData( hCombo,
            ComboBox_FindString( hCombo, 0, lpszDesc ),
            lpTemp );
    return( TRUE );
}
```

The enumeration is set in motion when the dialog containing the combo box is initialized:

```
if FAILED(DirectSoundEnumerate((LPDSENUMCALLBACK)DSEnumProc,
        &hCombo))
{
    EndDialog( hDlg, TRUE );
    return( TRUE );
}
```

In this case the address of the combo box handle is passed into **DirectSoundEnumerate**, which in turn passes it to the callback function. This parameter can be any 32-bit value that you want to have access to within the callback.

Creating the DirectSound Object

[This is preliminary documentation and subject to change.]

The simplest way to create the DirectSound object is with the **DirectSoundCreate** function. The first parameter of this function specifies the GUID of the device to be associated with the object. You can obtain this GUID by Enumeration of Sound Devices, or you can simply pass NULL to create the object for the default device.

```
LPDIRECTSOUND lpds;
HRESULT hr = DirectSoundCreate(NULL, &lpds, NULL));
```

The function returns an error if there is no sound device or if the sound device is under the control of an application using the waveform-audio (non-DirectSound) functions. You should prepare your applications for this call to fail so that they can either continue without sound or prompt the user to close the application that is already using the sound device.

You can also create the DirectSound object by using the **CoCreateInstance** function, as follows:

1. Initialize COM at the start of your application by calling **CoInitialize** and specifying NULL.

   ```
   if FAILED(CoInitialize(NULL))
       return FALSE;
   ```

2. Create your DirectSound object by using **CoCreateInstance** and the **IDirectSound::Initialize** method, rather than the **DirectSoundCreate** function.

   ```
   LPDIRECTSOUND lpds;
   dsrval = CoCreateInstance(&CLSID_DirectSound,
                             NULL,
                             CLSCTX_INPROC_SERVER,
                             &IID_IDirectSound,
                             &lpds);
   if SUCCEEDED(dsrval)
       dsrval = IDirectSound_Initialize(lpds, NULL);
   ```

CLSID_DirectSound is the class identifier of the DirectSound driver object class and *IID_IDirectSound* is the DirectSound interface that you should use. The *lpds* parameter is the uninitialized object **CoCreateInstance** returns.

Before you use a DirectSound object created with the **CoCreateInstance** function, you must call the **IDirectSound::Initialize** method. This method takes the same

driver GUID parameter that **DirectSoundCreate** uses (NULL in this case). After the DirectSound object is initialized, you can use and release the DirectSound object as if it had been created by using the **DirectSoundCreate** function.

Before you close the application, close the COM library by calling the **CoUninitialize** function, as follows:

```
CoUninitialize();
```

Cooperative Levels

[This is preliminary documentation and subject to change.]

Because Windows is a multitasking environment, more than one application may be working with a device driver at any one time. Through the use of cooperative levels, DirectX makes sure that each application does not gain access to the device in the wrong way or at the wrong time. Each DirectSound application has a cooperative level that determines the extent to which it is allowed to access the device.

After creating a DirectSound object, you must set the cooperative level for the device with the **IDirectSound::SetCooperativeLevel** method before you can play sounds.

The following example sets the cooperative level for the DirectSound device initialized at Creating the DirectSound Object. The *hwnd* parameter is the handle to the application window.

```
HRESULT hr = lpDirectSound->lpVtbl->SetCooperativeLevel(
        lpDirectSound, hwnd, DSSCL_NORMAL);
```

DirectSound defines four cooperative levels for sound devices: normal, priority, exclusive, and write-primary. Most applications will use the sound device at the primary cooperative level, which allows for orderly switching between applications that use the sound card but also makes it possible to set the primary buffer to 16-bit output.

Normal Cooperative Level

At the normal cooperative level, the application cannot set the format of the primary sound buffer, write to the primary buffer, or compact the on-board memory of the device. All applications at this cooperative level use a primary buffer format of 22 kHz, stereo sound, and 8-bit samples, so that the device can switch between applications as smoothly as possible.

Priority Cooperative Level

When using a DirectSound device with the priority cooperative level, the application has first rights to hardware resources, such as hardware mixing, and can set the format of the primary sound buffer and compact the on-board memory of the device.

Exclusive Cooperative Level

At the exclusive cooperative level, the application has all the privileges of the priority level. In addition, when the application is in the foreground, its buffers are the only ones that are audible.

Write-primary Cooperative Level

The highest cooperative level is write-primary. When using a DirectSound device with this cooperative level, your application has direct access to the primary sound buffer. In this mode, the application must write directly to the primary buffer. Secondary buffers cannot be played while this is happening.

An application must be set to the write-primary level in order to obtain direct write access to the audio samples in the primary buffer. If the application is not set to this level, then all calls to the **IDirectSoundBuffer::Lock** method for the primary buffer will fail.

When your application is set to the write-primary cooperative level and gains the foreground, all secondary buffers for other applications are stopped and marked as lost. When your application in turn moves to the background, its primary buffer is marked as lost and must be restored when the application again moves to the foreground. For more information, see Buffer Management.

You cannot set the write-primary cooperative level if a DirectSound driver is not present on the user's system. To determine whether this is the case, call the **IDirectSound::GetCaps** method and check for the DSCAPS_EMULDRIVER flag in the **DSCAPS** structure.

For more information, see Access to the Primary Buffer.

Device Capabilities

[This is preliminary documentation and subject to change.]

DirectSound allows your application to retrieve the hardware capabilities of the sound device. Most applications will not need to do this, because DirectSound automatically takes advantage of any available hardware acceleration. However, high-performance applications can use the information to scale their sound requirements to the available hardware. For example, an application might play more sounds if hardware mixing is available than if it is not.

After calling the **DirectSoundCreate** function to create a DirectSound object, your application can retrieve the capabilities of the sound device by calling the **IDirectSound::GetCaps** method.

The following example retrieves the capabilities of the device that was initialized in Creating the DirectSound Object.

```
DSCAPS dscaps;
```

```
dscaps.dwSize = sizeof(DSCAPS);
HRESULT hr = lpDirectSound->lpVtbl->GetCaps(lpDirectSound,
   &dscaps);
```

The **DSCAPS** structure receives information about the performance and resources of the sound device, including the maximum resources of each type and the resources that are currently available. Note that the **dwSize** member of this structure must be initialized before the method is called.

If your application scales to hardware capabilities, you should call the **IDirectSound::GetCaps** method between every buffer allocation to determine if there are enough resources to create the next buffer.

Speaker Configuration

[This is preliminary documentation and subject to change.]

The **IDirectSound** interface contains two methods that allow your application to investigate and set the configuration of the system's speakers; that is, their location relative to the listener. These methods are **IDirectSound::GetSpeakerConfig** and **IDirectSound::SetSpeakerConfig**.

DirectSound uses the speaker configuration to optimize 3-D effects for the user's sound system.

Compacting Hardware Memory

[This is preliminary documentation and subject to change.]

As long as it has at least the priority cooperative level, your application can use the **IDirectSound::Compact** method to move any on-board sound memory into a contiguous block to make the largest portion of free memory available.

DirectSound Buffers

[This is preliminary documentation and subject to change.]

This section covers the creation and management of DirectSoundBuffer objects, which are the fundamental mechanism for playing sounds. The following topics are discussed:

- Buffer Basics
- Static and Streaming Sound Buffers
- Creating Secondary Buffers
- Buffer Control Options
- Access to the Primary Buffer
- Playing Sounds

- Playback Controls
- Current Play and Write Positions
- Play Buffer Notification
- Mixing Sounds
- Custom Mixers
- Buffer Management
- Compressed Wave Formats

Most of the information in this section applies to 3-D sound buffers as well. For information specific to the **IDirectSound3DBuffer** interface, see DirectSound 3-D Buffers.

For information about capture buffers, see **DirectSoundCapture**.

Buffer Basics

[This is preliminary documentation and subject to change.]

When you initialize DirectSound in your application, it automatically creates and manages a primary sound buffer for mixing sounds and sending them to the output device.

Your application must create at least one secondary sound buffer for storing and playing individual sounds. For more information on how to do this, see Creating Secondary Buffers.

A secondary buffer can exist throughout the life of an application or it may be destroyed when no longer needed. It may contain a single sound that is to be played repeatedly, such as a sound effect in a game, or it may be filled with new data from time to time. The application can play a sound stored in a secondary buffer as a single event or as a looping sound that plays continuously. Secondary buffers can also be used to stream data, in cases where a sound file contains more data than can conveniently be stored in memory.

For more information on the different kinds of secondary buffers, see Static and Streaming Sound Buffers.

You can create two or more secondary buffers in the same physical memory by using the **IDirectSound::DuplicateSoundBuffer** method. Note that if the original buffer is in hardware memory and hardware resources are not available for the duplicate buffer, this call may fail.

You mix sounds from different secondary buffers simply by playing them at the same time. Any number of secondary buffers can be played at one time, up to the limits of available processing power.

The DirectSound mixer can provide as little as 20 milliseconds of latency, so there is no perceptible delay before play begins. Under these conditions, if your application plays a buffer and immediately begins a screen animation, the audio and video

appear to start at the same time. However, if DirectSound must emulate hardware features in software, the mixer cannot achieve low latency and a longer delay (typically 100-150 milliseconds) occurs before the sound is reproduced.

Normally you do not have to concern yourself at all with the primary buffer; DirectSound manages it behind the scenes. However, if your application is to perform its own mixing, DirectSound will let you write directly to the primary buffer. If you do this, you cannot also use secondary buffers. For more information, see Access to the Primary Buffer.

Static and Streaming Sound Buffers

[This is preliminary documentation and subject to change.]

When you create a secondary sound buffer, you specify whether it is a *static sound buffer* or a *streaming sound buffer*. A static buffer contains a complete sound in memory. A streaming buffer holds only a portion of a sound, such as 3 seconds of data from a 15-second bit of voice dialog. When using a streaming sound buffer, your application must periodically write new data to the buffer.

If a sound device has on-board sound memory, DirectSound attempts to place static buffers in the hardware memory. These buffers can then take advantage of hardware mixing, and the processing system incurs little or no overhead to mix these sounds. This is particularly useful for sounds your application plays repeatedly, because the sound data must be downloaded only once to the hardware memory.

Streaming buffers are generally located in main system memory to allow efficient writing to the buffer, although you can use hardware mixing on peripheral component interconnect (PCI) machines or other fast buses.

DirectSound distinguishes between static and streaming buffers in order to optimize performance, but it does not restrict how you can use the buffer. If a streaming buffer is big enough, there is nothing to prevent you from writing an entire sound to it in one chunk. In fact, if you do not intend to use the sound more than once, it can be more efficient to use a streaming buffer because by doing so you eliminate the step of downloading the data to hardware memory.

Your application may attempt to explicitly locate buffers in hardware or software. If you attempt to create a hardware buffer and there is insufficient memory or mixing capacity, the buffer creation request fails. Many existing sound cards do not have any on-board memory or mixing capacity, so no hardware buffers can be created on these devices.

For more information, see Creating Secondary Buffers.

Creating Secondary Buffers

[This is preliminary documentation and subject to change.]

To create a sound buffer, your application fills a **DSBUFFERDESC** structure and then passes its address to the **IDirectSound::CreateSoundBuffer** method. This

method creates a DirectSoundBuffer object and returns a pointer to an **IDirectSoundBuffer** interface. Your application uses this interface to manipulate and play the buffer.

The following example illustrates how to create a basic secondary sound buffer:

```
BOOL AppCreateBasicBuffer(
        LPDIRECTSOUND lpDirectSound,
        LPDIRECTSOUNDBUFFER *lplpDsb)
{
    PCMWAVEFORMAT pcmwf;
    DSBUFFERDESC dsbdesc;
    HRESULT hr;
    // Set up wave format structure.
    memset(&pcmwf, 0, sizeof(PCMWAVEFORMAT));
    pcmwf.wf.wFormatTag = WAVE_FORMAT_PCM;
    pcmwf.wf.nChannels = 2;
    pcmwf.wf.nSamplesPerSec = 22050;
    pcmwf.wf.nBlockAlign = 4;
    pcmwf.wf.nAvgBytesPerSec =
        pcmwf.wf.nSamplesPerSec * pcmwf.wf.nBlockAlign;
    pcmwf.wBitsPerSample = 16;
    // Set up DSBUFFERDESC structure.
    memset(&dsbdesc, 0, sizeof(DSBUFFERDESC)); // Zero it out.
    dsbdesc.dwSize = sizeof(DSBUFFERDESC);
    // Need default controls (pan, volume, frequency).
    dsbdesc.dwFlags = DSBCAPS_CTRLDEFAULT;
    // 3-second buffer.
    dsbdesc.dwBufferBytes = 3 * pcmwf.wf.nAvgBytesPerSec;
    dsbdesc.lpwfxFormat = (LPWAVEFORMATEX)&pcmwf;
    // Create buffer.
    hr = lpDirectSound->lpVtbl->CreateSoundBuffer(lpDirectSound,
        &dsbdesc, lplpDsb, NULL);
    if SUCCEEDED(hr)
    {
        // Valid interface is in *lplpDsb.
        return TRUE;
    }
    else
    {
        // Failed.
        *lplpDsb = NULL;
        return FALSE;
    }
}
```

Your application should create buffers for the most important sounds first, and then create buffers for other sounds in descending order of importance. DirectSound allocates hardware resources to the first buffer that can take advantage of them.

If your application must explicitly locate buffers in hardware or software, you can specify either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag in the **DSBUFFERDESC** structure. If the DSBCAPS_LOCHARDWARE flag is specified and there is insufficient hardware memory or mixing capacity, the buffer creation request fails.

You can ascertain the location of an existing buffer by using the **IDirectSoundBuffer::GetCaps** method and checking the **dwFlags** member of the **DSBCAPS** structure for either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags. One or the other is always specified.

When you create a sound buffer, you can indicate that a buffer is static by specifying the DSBCAPS_STATIC flag. If you do not specify this flag, the buffer is a streaming buffer. For more information, see Static and Streaming Sound Buffers.

DirectSoundBuffer objects are owned by the DirectSound object that created them. When the DirectSound object is released, all buffers created by that object also will be released and should not be referenced.

Buffer Control Options

[This is preliminary documentation and subject to change.]

When creating a sound buffer, your application must specify the control options needed for that buffer. This is done with the **dwFlags** member of the **DSBUFFERDESC** structure, which can contain one or more DSBCAPS_CTRL* flags. The following controls are available:

- 3-D properties
- Frequency
- Pan
- Volume
- Position notification

To obtain the best performance on all sound cards, your application should specify only control options it will use.

DirectSound uses the control options in determining whether hardware resources can be allocated to sound buffers. For example, a device might support hardware buffers but provide no pan control on those buffers. In this case, DirectSound would use hardware acceleration only if the DSBCAPS_CTRLPAN flag was not specified.

If your application attempts to use a control that a buffer lacks, the method call fails. For example, if you attempt to change the volume by using the **IDirectSoundBuffer::SetVolume** method, the method can succeed only if the DSBCAPS_CTRLVOLUME flag was specified when the buffer was created.

Otherwise the method fails and returns the DSERR_CONTROLUNAVAIL error code.

Access to the Primary Buffer

[This is preliminary documentation and subject to change.]

For applications that require specialized mixing or other effects not supported by secondary buffers, DirectSound allows direct access to the primary buffer.

When you obtain write access to a primary sound buffer, other DirectSound features become unavailable. Secondary buffers are not mixed and, consequently, hardware-accelerated mixing is unavailable.

Most applications should use secondary buffers instead of directly accessing the primary buffer. Applications can write to a secondary buffer easily because the larger buffer size provides more time to write the next block of data, thereby minimizing the risk of gaps in the audio. Even if an application has simple audio requirements, such as using one stream of audio data that does not require mixing, it will achieve better performance by using a secondary buffer to play its audio data.

You cannot specify the size of the primary buffer, and you must accept the returned size after the buffer is created. A primary buffer is typically very small, so if your application writes directly to this kind of buffer, it must write blocks of data at short intervals to prevent the previously written data from being replayed.

You create an accessible primary buffer by specifying the DSBCAPS_PRIMARYBUFFER flag in the **DSBUFFERDESC** structure passed to the **IDirectSound::CreateSoundBuffer** method. If you want to write to the buffer, the cooperative level must be DSSCL_WRITEPRIMARY.

You cannot obtain write access to a primary buffer unless it exists in hardware. To determine whether this is the case, call the **IDirectSoundBuffer::GetCaps** method and check for the DSBCAPS_LOCHARDWARE flag in the **dwFlags** member of the **DSBCAPS** structure that is returned. If you attempt to lock a primary buffer that is emulated in software, the call will fail.

Primary sound buffers must be played with looping. Ensure that the DSBPLAY_LOOPING flag is set.

The following example shows how to obtain write access to the primary buffer:

```
BOOL AppCreateWritePrimaryBuffer(
    LPDIRECTSOUND lpDirectSound,
    LPDIRECTSOUNDBUFFER *lplpDsb,
    LPDWORD lpdwBufferSize,
    HWND hwnd)
{
    DSBUFFERDESC dsbdesc;
    DSBCAPS dsbcaps;
    HRESULT hr;
```

```
WAVEFORMATEX wf;

// Set up wave format structure.
memset(&wf, 0, sizeof(WAVEFORMATEX));
wf.wFormatTag = WAVE_FORMAT_PCM;
wf.nChannels = 2;
wf.nSamplesPerSec = 22050;
wf.nBlockAlign = 4;
wf.nAvgBytesPerSec =
   wf.nSamplesPerSec * wf.nBlockAlign;
wf.wBitsPerSample = 16;

// Set up DSBUFFERDESC structure.
memset(&dsbdesc, 0, sizeof(DSBUFFERDESC));
dsbdesc.dwSize = sizeof(DSBUFFERDESC);
dsbdesc.dwFlags = DSBCAPS_PRIMARYBUFFER;
// Buffer size is determined by sound hardware.
dsbdesc.dwBufferBytes = 0;
dsbdesc.lpwfxFormat = NULL; // Must be NULL for primary buffers.

// Obtain write-primary cooperative level.
hr = lpDirectSound->lpVtbl->SetCooperativeLevel(lpDirectSound,
   hwnd, DSSCL_WRITEPRIMARY);
if SUCCEEDED(hr)
{
   // Try to create buffer.
   hr = lpDirectSound->lpVtbl->CreateSoundBuffer(lpDirectSound,
      &dsbdesc, lplpDsb, NULL);
   if SUCCEEDED(hr)
   {
      // Set primary buffer to desired format.
      hr = (*lplpDsb)->lpVtbl->SetFormat(*lplpDsb, &wf);
      if SUCCEEDED(hr)
      {
         // If you want to know the buffer size, call GetCaps.
            dsbcaps.dwSize = sizeof(DSBCAPS);
         (*lplpDsb)->lpVtbl->GetCaps(*lplpDsb, &dsbcaps);
         *lpdwBufferSize = dsbcaps.dwBufferBytes;
         return TRUE;
      }
   }
}
// Failure.
*lplpDsb = NULL;
*lpdwBufferSize = 0;
return FALSE;
```

```
}
```

You may also create a primary buffer object without write access, by specifying a cooperative level other than DSSCL_WRITEPRIMARY. One reason for doing this would be to call the **IDirectSoundBuffer::Play** method for the primary buffer, in order to eliminate problems associated with frequent short periods of silence. For more information, see Playing the Primary Buffer Continuously.

See also Custom Mixers.

Playing Sounds

[This is preliminary documentation and subject to change.]

Playing a sound consists of the following steps:

1. Lock a portion of the secondary buffer using **IDirectSoundBuffer::Lock**. This method returns a pointer to the address where writing will begin, based on the offset from the beginning of the buffer that you pass in to the method.
2. Write the audio data to the buffer.
3. Unlock the buffer using **IDirectSoundBuffer::Unlock**.
4. Send the sound to the primary buffer and from there to the output device using **IDirectSoundBuffer::Play**. If the buffer is a streaming buffer, it will continue playing in a loop as steps 1 to 3 are repeated.

Because streaming sound buffers usually play continually and are conceptually circular, DirectSound returns two write pointers when locking a sound buffer. For example, if you tried to lock 3,000 bytes beginning at the midpoint of a 4,000-byte buffer, the **Lock** method would return one pointer to the last 2,000 bytes of the buffer, and a second pointer to the first 1,000 bytes. The second pointer is NULL if the locked portion of the buffer does not wrap around.

Normally the buffer stops playing automatically when the end is reached. However, if the DSBPLAY_LOOPING flag was set in the *dwFlags* parameter to the **Play** method, the buffer plays repeatedly until the application calls the **IDirectSoundBuffer::Stop** method.

For streaming sound buffers, your application is responsible for ensuring that each block of data is written to the buffer ahead of the current play position. (For more on the play position, see Current Play and Write Positions.) Applications should write at least 1 second ahead of the current play position to minimize the possibility of gaps in the audio output during playback.

The following C example writes data to a sound buffer, starting at the offset into the buffer passed in *dwOffset*:

```
BOOL AppWriteDataToBuffer(
    LPDIRECTSOUNDBUFFER lpDsb,  // the DirectSound buffer
    DWORD dwOffset,             // our own write cursor
```

```
    LPBYTE lpbSoundData,        // start of our data
    DWORD dwSoundBytes)         // size of block to copy
{
    LPVOID  lpvPtr1;
    DWORD   dwBytes1;
    LPVOID  lpvPtr2;
    DWORD   dwBytes2;
    HRESULT hr;

    // Obtain memory address of write block. This will be in two parts
    // if the block wraps around.
    hr = lpDsb->lpVtbl->Lock(lpDsb, dwOffset, dwSoundBytes, &lpvPtr1,
        &dwBytes1, &lpvPtr2, &dwBytes2, 0);

    // If DSERR_BUFFERLOST is returned, restore and retry lock.
    if (DSERR_BUFFERLOST == hr)
    {
        lpDsb->lpVtbl->Restore(lpDsb);
        hr = lpDsb->lpVtbl->Lock(lpDsb, dwOffset, dwSoundBytes,
            &lpvPtr1, &dwAudio1, &lpvPtr2, &dwAudio2, 0);
    }
    if SUCCEEDED(hr)
    {
        // Write to pointers.
        CopyMemory(lpvPtr1, lpbSoundData, dwBytes1);
        if (NULL != lpvPtr2)
        {
            CopyMemory(lpvPtr2, lpbSoundData+dwBytes1, dwBytes2);
        }
        // Release the data back to DirectSound.
        hr = lpDsb->lpVtbl->Unlock(lpDsb, lpvPtr1, dwBytes1, lpvPtr2,
            dwBytes2);
        if SUCCEEDED(hr)
        {
            // Success.
            return TRUE;
        }
    }
    // Lock, Unlock, or Restore failed.
    return FALSE;
}
```

Playback Controls

[This is preliminary documentation and subject to change.]

To retrieve and set the volume at which a buffer is played, your application can use the **IDirectSoundBuffer::GetVolume** and **IDirectSoundBuffer::SetVolume** methods. Setting the volume on the primary sound buffer changes the waveform-audio volume of the sound card.

Similarly, by calling the **IDirectSoundBuffer::GetFrequency** and **IDirectSoundBuffer::SetFrequency** methods, you can retrieve and set the frequency at which audio samples play. You cannot change the frequency of the primary buffer.

To retrieve and set the pan, you can call the **IDirectSoundBuffer::GetPan** and **IDirectSoundBuffer::SetPan** methods. You cannot change the pan of the primary buffer.

In order to use any of these controls, you must have set the appropriate flags when creating the buffer. See Buffer Control Options.

Current Play and Write Positions

[This is preliminary documentation and subject to change.]

DirectSound maintains two pointers into the buffer: the current play position (sometimes called the play cursor) and the current write position (or write cursor). These positions are byte offsets into the buffer, not absolute memory addresses.

The **IDirectSoundBuffer::Play** method always starts playing at the buffer's current play position. When a buffer is created, the play position is set to zero. As a sound is played, the play position moves and always points to the next byte of data to be output. When the buffer is stopped, the play position remains at the next byte of data.

The current write position is the point after which it is safe to write data into the buffer. The block between the current play position and the current write position is already committed to be played, and cannot be changed safely.

Visualize the buffer as a clock face, with data written to it in a clockwise direction. The play position and the write position are like two hands sweeping around the face at the same speed, the write position always keeping a little ahead of the play position. If the play position points to the 1 and the write position points to the 2, it is only safe to write data after the 2. Data between the 1 and the 2 may already have been queued for playback by DirectSound and should not be touched.

Note

The write position moves with the play position, not with data written to the buffer. If you're streaming data, you are responsible for maintaining your own pointer into the buffer to indicate where the next block of data should be written. Also note that the *dwWriteCursor* parameter to the **IDirectSoundBuffer::Lock** method is not the current write position; it is the offset within the buffer where you actually intend to begin writing data. (If you do want to begin writing at the

current write position, you specify DSBLOCK_FROMWRITECURSOR in the *dwFlags* parameter. In this case the *dwWriteCursor* parameter is ignored.)

An application can retrieve the current play and write positions by calling the **IDirectSoundBuffer::GetCurrentPosition** method. The **IDirectSoundBuffer::SetCurrentPosition** method lets you set the current play position, but the current write position cannot be changed.

To ensure that the current play position is reported as accurately as possible, you should always specify the DSBCAPS_GETCURRENTPOSITION2 flag when creating a secondary buffer. For more information, see **DSBUFFERDESC**.

Play Buffer Notification

[This is preliminary documentation and subject to change.]

When streaming audio, you may want your application to be notified when the current play position reaches a certain point in the buffer, or when playback is stopped. With the **IDirectSoundNotify::SetNotificationPositions** method you can set any number of points within the buffer where events are to be signaled. You cannot do this while the buffer is playing.

First you have to obtain a pointer to the **IDirectSoundNotify** interface. You can do this by using the buffer object's **QueryInterface** method, as in the following C++ example:

```
// LPDIRECTSOUNDBUFFER lpDsbSecondary;
// The buffer has been initialized already.

LPDIRECTSOUNDNOTIFY lpDsNotify;  // pointer to the interface

HRESULT hr = lpDsbSecondary->QueryInterface(IID_IDirectSoundNotify,
                               (LPVOID *)&lpDsNotify);
if SUCCEEDED(hr)
{
   // Go ahead and use lpDsNotify->SetNotificationPositions.
}
```

Note

The **IDirectSoundNotify** interface is associated with the object that obtained the pointer, in this case the secondary buffer. The methods of the new interface will automatically apply to that buffer.

Now create an event object with the Win32® **CreateEvent** function. You put the handle to this event in the **hEventNotify** member of a **DSBPOSITIONNOTIFY** structure, and in the **dwOffset** member of that structure you specify the offset within the buffer where you want the event to be signaled. Then you pass the address of the structure—or of an array of structures, if you want to set more than one notification position—to the **IDirectSoundNotify::SetNotificationPositions** method.

The following example sets a single notification position. The event will be signaled when playback stops, either because it was not looping and the end of the buffer has been reached, or because the application called the **IDirectSoundBuffer::Stop** method.

```
DSBPOSITIONNOTIFY PositionNotify;

PositionNotify.Offset = DSBPN_OFFSETSTOP;
PositionNotify.hEventNotify = hMyEvent;
// hMyEvent is the handle returned by CreateEvent()

lpDsNotify->SetNotificationPositions(1, &PositionNotify);
```

Mixing Sounds

[This is preliminary documentation and subject to change.]

It is easy to mix multiple streams with DirectSound. You simply create secondary sound buffers, obtaining an **IDirectSoundBuffer** interface for each sound. You then play the buffers simultaneously. DirectSound takes care of the mixing in the primary sound buffer and plays the result.

The DirectSound mixer can obtain the best results from hardware acceleration if your application correctly specifies the DSBCAPS_STATIC flag for static buffers. This flag should be specified for any static buffers that will be reused. DirectSound downloads these buffers to the sound hardware memory, where available, and consequently does not incur any processing overhead in mixing these buffers. The most important static sound buffers should be created first to give them first priority for hardware acceleration.

The DirectSound mixer produces the best sound quality if all your application's sounds use the same wave format and the hardware output format is matched to the format of the sounds. If this is done, the mixer need not perform any format conversion.

Your application can change the hardware output format by creating a primary sound buffer and calling the **IDirectSoundBuffer::SetFormat** method. Note that this primary buffer is for control purposes only; creating it is not the same as obtaining write access to the primary buffer as described under Access to the Primary Buffer, and you do not need the DSSCL_WRITEPRIMARY cooperative level. However, you do need a cooperative level of DSSCL_PRIORITY or higher in order to call the **SetFormat** method. DirectSound will restore the hardware format to the format specified in the last call every time the application gains the input focus.

Custom Mixers

[This is preliminary documentation and subject to change.]

Most applications will use the DirectSound mixer; it should be sufficient for almost all mixing needs and it automatically takes advantage of any available hardware acceleration. However, if an application requires some other functionality that DirectSound does not provide, it can obtain write access to the primary sound buffer and mix streams directly into it.

To implement a custom mixer, the application must first set the DSSCL_WRITEPRIMARY cooperative level and then create a primary sound buffer. (See Access to the Primary Buffer.) It can then lock the buffer, write data to it, unlock it, and play it just like any other buffer. (See Playing Sounds.) Note however that the DSBPLAY_LOOPING flag must be specified or the **IDirectSoundBuffer::Play** call will fail.

The following example illustrates how an application might implement a custom mixer. The AppMixIntoPrimaryBuffer sample function would have to be called at regular intervals, frequently enough to prevent the sound device from repeating blocks of data. The CustomMixer function is an application-defined function that mixes several streams together, as specified in the application-defined AppStreamInfo structure, and writes the result to the specified pointer.

```
BOOL AppMixIntoPrimaryBuffer(
        LPAPPSTREAMINFO lpAppStreamInfo,
        LPDIRECTSOUNDBUFFER lpDsbPrimary,
        DWORD dwDataBytes,
        DWORD dwOldPos,
        LPDWORD lpdwNewPos)
{
    LPVOID lpvPtr1;
    DWORD dwBytes1;
    LPVOID lpvPtr2;
    DWORD dwBytes2;
    HRESULT hr;
    // Obtain write pointer.
    hr = lpDsbPrimary->lpVtbl->Lock(lpDsbPrimary,
                              dwOldPos, dwDataBytes,
                              &lpvPtr1, &dwBytes1,
                              &lpvPtr2, &dwBytes2, 0);
    // If DSERR_BUFFERLOST is returned, restore and retry lock.
    if (DSERR_BUFFERLOST == hr)
    {
        lpDsbPrimary->lpVtbl->Restore(lpDsbPrimary);
        hr = lpDsbPrimary->lpVtbl->Lock(lpDsbPrimary,
                                  dwOldPos, dwDataBytes,
                                  &lpvPtr1, &dwBytes1,
                                  &lpvPtr2, &dwBytes2, 0);
    }
    if SUCCEEDED(hr)
    {
```

```
        // Mix data into the returned pointers.
        CustomMixer(lpAppStreamInfo, lpvPtr1, dwBytes1);
        *lpdwNewPos = dwOldPos + dwBytes1;
        if (NULL != lpvPtr2)
        {
            CustomMixer(lpAppStreamInfo, lpvPtr2, dwBytes2);
            *lpdwNewPos = dwBytes2; // Because it wrapped around.
        }
        // Release the data back to DirectSound.
        hr = lpDsbPrimary->lpVtbl->Unlock(lpDsbPrimary,
                                lpvPtr1, dwBytes1,
                                lpvPtr2, dwBytes2);
        if SUCCEEDED(hr)
        {
            return TRUE;
        }
    }
    // Lock or Unlock failed.
    return FALSE;
}
```

Buffer Management

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::GetCaps** method retrieves the capabilities of the DirectSoundBuffer object.

Your application can use the **IDirectSoundBuffer::GetStatus** method to determine if the current sound buffer is playing or if it has stopped.

You can use the **IDirectSoundBuffer::GetFormat** method to retrieve information about the format of the sound data in the buffer. You also can use the **IDirectSoundBuffer::GetFormat** and **IDirectSoundBuffer::SetFormat** methods to retrieve and set the format of the sound data in the primary sound buffer.

Note

After a secondary sound buffer is created, its format is fixed. If you need a secondary buffer that uses another format, you must create a new sound buffer with this format.

Memory for a sound buffer can be lost in certain situations: for example, when buffers are located in sound card memory and another application gains control of the hardware resources. Loss can also occur when an application with the write-primary cooperative level moves to the foreground; in this case, DirectSound makes all other sound buffers lost so that the foreground application can write directly to the primary buffer.

The DSERR_BUFFERLOST error code is returned when the **IDirectSoundBuffer::Lock** or **IDirectSoundBuffer::Play** method is called for a lost buffer. When the application that caused the loss either lowers its cooperative level from write-primary or moves to the background, other applications can attempt to reallocate the buffer memory by calling the **IDirectSoundBuffer::Restore** method. If successful, this method restores the buffer memory and all other settings for the buffer, such as volume and pan settings. However, a restored buffer may not contain valid sound data, so the owning application should rewrite the data to the buffer.

Compressed Wave Formats

[This is preliminary documentation and subject to change.]

DirectSound does not currently support compressed wave formats. Applications should use the audio compression manager (ACM) functions, provided with the Win32 APIs in the Platform SDK, to convert compressed audio to pulse-code modulation (PCM) format before writing the data to a sound buffer. In fact, by locking a pointer to the sound-buffer memory and passing this pointer to the ACM, the data can be decoded directly to the sound buffer for maximum efficiency.

DirectSound in 3-D

[This is preliminary documentation and subject to change.]

DirectSound enables an application to change the apparent position and orientation of a sound source or listener, and also to suggest the relative velocity of the source and listener by using Doppler shift.

The following topics cover some general aspects of 3-D sound:

- Integration with Direct3D
- Mono and Stereo Sources
- Perception of Sound Positions

Information on how to use 3-D sound in an application is found in the following sections:

- DirectSound 3-D Buffers
- DirectSound 3-D Listeners

Integration with Direct3D

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer** and **IDirectSound3DListener** interfaces are designed to work together with Direct3D®. The positioning information used by Direct3D to arrange objects in a virtual environment can also be used to arrange sound sources.

The **D3DVECTOR** and **D3DVALUE** types that are familiar to Direct3D programmers are also used in the **IDirectSound3DBuffer** and **IDirectSound3DListener** interfaces. The same left-handed coordinate system used by Direct3D is employed by DirectSound. (For information about coordinate systems, see 3-D Coordinate Systems, in the Direct3D overview material.)

You can use the system callback mechanism of Direct3D to simplify the implementation of 3-D sound in your application. For example, you could use the **D3DRMFRAMEMOVECALLBACK** application-defined function to monitor the movement of a frame in an application and change the sonic environment only when a certain condition has been reached.

In DirectSound, distances are normally measured in meters. If your application does not use the meter as its unit of measure for 3-D graphics, you can set a distance factor, which is a floating-point value representing meters per application-specified distance unit. For example, if your application uses feet, it could specify a DirectSound distance factor of .3048, which is the number of meters in a foot.

Mono and Stereo Sources

[This is preliminary documentation and subject to change.]

Stereo sound sources are not particularly useful in the 3-D sound environments of DirectSound, because DirectSound creates its own stereo output from a monaural input. If an application uses stereo sound buffers, the left and right values for each sample are averaged before the 3-D processing is applied.

Applications should supply monaural sound sources when using the 3-D capabilities of DirectSound. Although the system can convert a stereo source into mono, there is no reason to supply stereo, and the conversion step wastes time.

Perception of Sound Positions

[This is preliminary documentation and subject to change.]

In the real world, the perception of a sound's position in space is influenced by a number of factors, including the following:

- *Volume.* The farther an object is from the listener, the quieter it sounds. This phenomenon is known as rolloff.
- *Interaural intensity difference*. A sound coming from the listener's right will sound louder in the right ear than in the left. This effect is familiar to anyone who has listened to a stereo sound system.
- *Interaural time difference.* A sound emitted by a source to the listener's right will arrive at the right ear slightly before it arrives at the left ear. (The duration of this offset is approximately a millisecond.)
- *Muffling.* The orientation of the ears ensures that sounds coming from behind the listener are slightly muffled compared with sounds coming from in front. In

addition, if a sound is coming from the right, the sound reaching the left ear will be muffled by the mass of the listener's head as well as by the orientation of the left ear.

Although these are not the only cues people use to discern the position of sound, they are the main ones, and they are the factors that have been implemented in the positioning system of DirectSound. Hardware optimized for 3-D sound can support other cues as well, such as the effect of the earlobes on the pitch and timing of sounds arriving from different directions. The mathematics behind this effect are known as the *head-related transfer function*.

One of the most important sound-positioning cues, however, is still the apparent visual position of the sound source. If a projectile appears as a dot in the distance and grows to the size of an intercontinental missile before it roars past the viewer's head, the listener does not need subtle acoustical cues in order to sense the position and velocity of the sound source.

DirectSound 3-D Buffers

[This is preliminary documentation and subject to change.]

A 3-D sound buffer is created and managed like any other sound buffer, and all the methods of the **IDirectSoundBuffer** interface are available. However, in order to set 3-D parameters you need to obtain the **IDirectSound3DBuffer** interface for the buffer. This interface is supported only by sound buffers created with the DSBCAPS_CTRL3D flag.

This section describes how your applications can manage buffers with the **IDirectSound3DBuffer** interface methods. The following topics are discussed:

- Obtaining the IDirectSound3DBuffer Interface
- Batch Parameters for IDirectSound3DBuffer
- Minimum and Maximum Distances
- Processing Mode
- Buffer Position and Velocity
- Sound Cones

Obtaining the IDirectSound3DBuffer Interface

[This is preliminary documentation and subject to change.]

To obtain a pointer to an **IDirectSound3DBuffer** interface, you must first create a secondary 3-D sound buffer. Do this by using the **IDirectSound::CreateSoundBuffer** method, specifying the DSBCAPS_CTRL3D flag in the **dwFlags** member of the **DSBUFFERDESC** structure parameter. Then,

use the **IDirectSoundBuffer::QueryInterface** method on the resulting buffer to obtain a pointer to an **IDirectSound3DBuffer** interface for that buffer.

The following example calls the **QueryInterface** method with the C++ syntax:

```
// LPDIRECTSOUNDBUFFER lpDsbSecondary;
// The buffer has been created with DSBCAPS_CTRL3D.
LPDIRECTSOUND3DBUFFER lpDs3dBuffer;

HRESULT hr = lpDsbSecondary->QueryInterface(IID_IDirectSound3DBuffer,
                               (LPVOID *)&lpDs3dBuffer);
if SUCCEEDED(hr)
{
   // Set 3-D parameters of this sound.
   .
   .
   .
}
```

Note

Pan control conflicts with 3-D processing. If both DSBCAPS_CTRL3D and DSBCAPS_CTRLPAN are specified when the buffer is created, DirectSound returns an error.

Batch Parameters for IDirectSound3DBuffer

[This is preliminary documentation and subject to change.]

Applications can retrieve or set a 3-D sound buffer's parameters individually or in batches. To set individual values, your application can use the applicable **IDirectSound3DBuffer** interface method. However, applications often must set or retrieve all the values at once. You can do this with the **IDirectSound3DBuffer::GetAllParameters** and **IDirectSound3DBuffer::SetAllParameters** methods.

Minimum and Maximum Distances

[This is preliminary documentation and subject to change.]

As a listener approaches a sound source, the sound gets louder: the volume doubles when the distance is halved. Past a certain point, however, it is not reasonable for the volume to continue to increase. This is the *minimum distance* for the sound source.

The minimum distance is especially useful when an application must compensate for the difference in absolute volume levels of different sounds. Although a jet engine is much louder than a bee, for example, for practical reasons these sounds must be

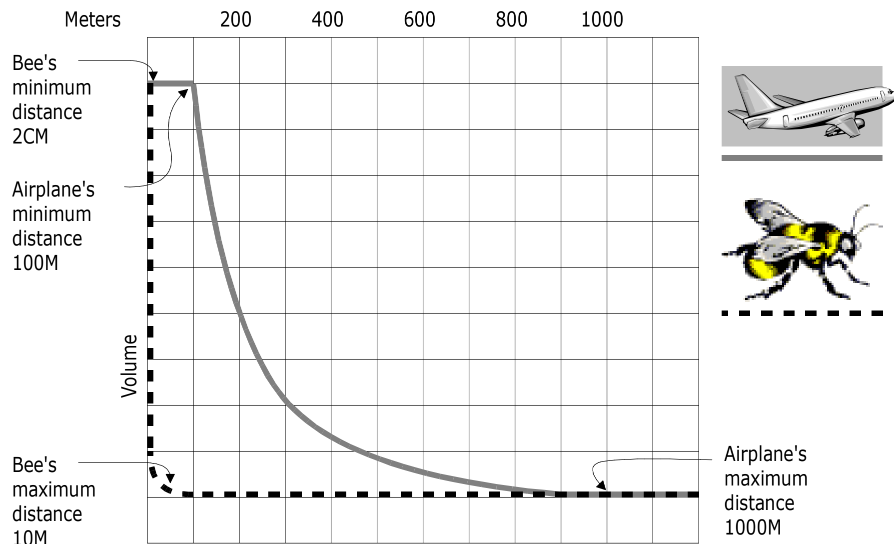
recorded at similar absolute volumes. An application might use a minimum distance of 100 meters for the jet engine and 2 centimeters for the bee. With these settings, the jet engine would be at half volume when the listener was 200 meters away, but the bee would be at half volume when the listener was 4 centimeters away.

The default minimum distance for a sound buffer, DS3D_DEFAULTMINDISTANCE, is currently defined as 1 unit. Unless you change this value, the sound will be only half as loud when it is 2 meters away from the user, a quarter as loud at 4 meters, and so on. For most sounds you will probably want to set a larger minimum distance.

The *maximum distance* for a sound source is the distance beyond which the sound does not get any quieter. The default maximum distance for a DirectSound 3-D buffer (DS3D_DEFAULTMAXDISTANCE) is a huge number, meaning that in most cases the attenuation will continue to be calculated even when the sound ceases to be audible. In order to avoid unnecessary processing, applications should set a reasonable maximum distance and include the DSBCAPS_MUTE3DATMAXDISTANCE flag when creating the buffer.

The maximum distance can also be used to prevent a sound from becoming inaudible. For example, if you have set the minimum distance for a sound at 100 meters, that sound might become effectively inaudible at 1,000 meters or less. By setting the maximum distance at 800 meters you would ensure that the sound always had at least one-eighth of its maximum volume regardless of the distance. In this case, of course, you would not set the DSBCAPS_MUTE3DATMAXDISTANCE flag.

The following illustration shows the concepts of minimum and maximum distance.



An application sets and retrieves the minimum distance value by using the **IDirectSound3DBuffer::SetMinDistance** and

IDirectSound3DBuffer::GetMinDistance methods. Similarly, it can set and retrieve the maximum distance value by using the **IDirectSound3DBuffer::SetMaxDistance** and **IDirectSound3DBuffer::GetMaxDistance** methods.

By default, distance values are expressed in meters. See Distance Factor.

To adjust the effect of distance on volume for all sound buffers, you can change the Rolloff Factor.

Processing Mode

[This is preliminary documentation and subject to change.]

Sound buffers have three processing modes: normal, head-relative, and disabled.

In normal mode, the sound source is positioned and oriented absolutely in world space. This is the default mode.

In head-relative mode, the buffer is automatically repositioned in world space as the listener moves and turns. Values set and retrieved through methods such as **IDirectSound3DBuffer::SetPosition**, **IDirectSound3DBuffer::SetVelocity**, and **IDirectSound3DBuffer::GetConeOrientation** are all relative to the current position, velocity, and orientation of the listener.

In disabled mode, 3-D sound processing is disabled and the sound seems to originate from the center of the listener's head.

An application sets the mode for a 3-D sound buffer by using the **IDirectSound3DBuffer::SetMode** method.

Buffer Position and Velocity

[This is preliminary documentation and subject to change.]

An application can set and retrieve a sound source's position in 3-D space by using the **IDirectSound3DBuffer::SetPosition** and **IDirectSound3DBuffer::GetPosition** methods. A position is expressed as a vector, relative to either world space or the listener, depending on the processing mode.

To set or retrieve the velocity value that DirectSound uses to calculate Doppler-shift effects for a listener, you use the **IDirectSound3DBuffer::SetVelocity** and **IDirectSound3DBuffer::GetVelocity** methods. Velocity is measured in distance units per second—by default, meters per second.

Note that velocity is completely independent of the actual position or movement of a buffer. It is entirely up to the application to set the appropriate velocity for a buffer.

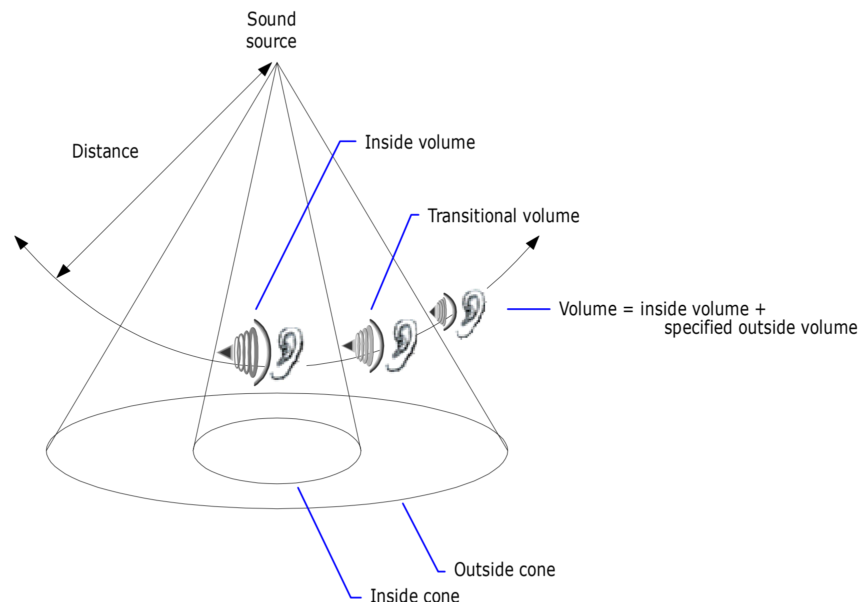
Sound Cones

[This is preliminary documentation and subject to change.]

A sound with no orientation is a point source; the amplitude of the sound at a given distance is the same in all directions. A sound with an orientation is a sound cone.

In DirectSound, sound cones are made up of an *inside cone* and an *outside cone*. Within the inside cone, the volume of the sound is just what it would be if there were no cone; in other words, there is no attenuation from the normal volume. (Of course, normal volume is not necessarily maximum volume. It has already been modified by the basic volume of the buffer — as set by the **IDirectSoundBuffer::SetVolume** method — as well as by the distance from the listener, the listener's orientation, the rolloff factor, and the buffer's minimum distance value.) Outside the outside cone, the normal volume is attenuated by a specified number of decibels, as set by the application. The angle between the inside and outside cones is a zone of transition from the inside volume to the outside volume.

The following illustration shows the concept of sound cones.



Technically, any 3-D sound buffer in DirectSound is a sound cone, but by default a buffer behaves like an omnidirectional sound source, because the outside volume is 0 (that is, there is no attenuation), and the inside and outside cone angles are 360 degrees. Until the application changes these values, the sound will not have any apparent orientation.

Designing sound cones properly can add dramatic effects to your application. For example, you could position a sound source in the center of a room, setting its orientation toward an open door in a hallway. Then set the angle of the inside cone so that it extends to the width of the doorway, make the outside cone a bit wider, and set the outside cone volume to inaudible. A listener moving along the hallway will begin to hear the sound when near the doorway, and the sound will be loudest as the listener passes in front of the open door.

An application sets or retrieves the angles that define sound cones by using the **IDirectSound3DBuffer::SetConeAngles** and **IDirectSound3DBuffer::GetConeAngles** methods. The outside cone angle must always be equal to or greater than the inside cone angle.

To set or retrieve the orientation of sound cones, an application can use the **IDirectSound3DBuffer::SetConeOrientation** and **IDirectSound3DBuffer::GetConeOrientation** methods.

An application sets and retrieves the outside cone volume by using the **IDirectSound3DBuffer::SetConeOutsideVolume** and **IDirectSound3DBuffer::GetConeOutsideVolume** methods. The outside cone volume is expressed in hundredths of decibels and is a negative value, because it represents attenuation from the default volume of 0.

DirectSound 3-D Listeners

[This is preliminary documentation and subject to change.]

A sound is only a sound when it is heard. The 3-D sound effects in a DirectSound application are affected not only by the position, orientation, and velocity of the sound buffer, but also by the position, orientation, and velocity of the listener.

The **IDirectSound3DListener** interface controls the listener's position, orientation, and apparent velocity in 3-D space. It also controls the general parameters of the acoustic environment, such as the amount of Doppler shift and the rate of volume attenuation over distance.

This section describes how your application can obtain a pointer to an **IDirectSound3DListener** interface and manage listener parameters by using interface methods. The following topics are discussed:

- Obtaining the IDirectSound3DListener Interface
- Batch Parameters for IDirectSound3DListener
- Deferred Settings
- Distance Factor
- Listener Orientation
- Listener Position and Velocity
- Doppler Factor
- Rolloff Factor

Obtaining the IDirectSound3DListener Interface

[This is preliminary documentation and subject to change.]

To obtain a pointer to an **IDirectSound3DListener** interface, you must first create a primary 3-D sound buffer. Do this by using the **IDirectSound::CreateSoundBuffer** method, specifying the DSBCAPS_CTRL3D and DSBCAPS_PRIMARYBUFFER flags in the **dwFlags** member of the accompanying **DSBUFFERDESC** structure. Then, use the **IDirectSoundBuffer::QueryInterface** method on the resulting buffer to obtain a pointer to an **IDirectSound3DListener** interface for that buffer, as shown in the following example with C++ syntax:

```
/* In this example, it is assumed that lpds is a valid
   pointer to a DirectSound object */

DSBUFFERDESC        dsbd;
LPDIRECTSOUNDBUFFER lpdsbPrimary;

ZeroMemory(&dsbd, sizeof(DSBUFFERDESC));
dsbd.dwSize = sizeof(DSBUFFERDESC);
dsbd.dwFlags = DSBCAPS_CTRL3D | DSBCAPS_PRIMARYBUFFER;
if SUCCEEDED(lpds->CreateSoundBuffer(&dsbd, &lpdsbPrimary, NULL))
{
    // Get listener interface
    if FAILED(lpdsbPrimary->QueryInterface(IID_IDirectSound3DListener,
            (LPVOID *)&lp3DListener))
    {
    lpdsbPrimary->Release();
    }
}
```

Batch Parameters for IDirectSound3DListener

[This is preliminary documentation and subject to change.]

Applications can retrieve or set a 3-D listener object's parameters individually or in batches. To set individual values, your application can use the applicable **IDirectSound3DListener** interface method. However, applications often must set or retrieve all the values that describe the listener at once. An application can perform these batch parameter manipulations in a single call by using the **IDirectSound3DListener::GetAllParameters** and **IDirectSound3DListener::SetAllParameters** methods.

Deferred Settings

[This is preliminary documentation and subject to change.]

Every change to 3-D sound buffer and listener settings causes DirectSound to remix, at the expense of CPU cycles. To minimize the performance impact of changing 3-D

settings, use the DS3D_DEFERRED flag in the *dwApply* parameter of any of the **IDirectSound3DListener** or **IDirectSound3DBuffer** methods that change 3-D settings. Then call the **IDirectSound3DListener::CommitDeferredSettings** method to execute all of the deferred commands at once.

Note

Any deferred settings are overwritten if your application calls the same setting method with the DS3D_IMMEDIATE flag before it calls **CommitDeferredSettings**. For example, if you set the listener velocity to (1.0, 0.0, 0.0) with the DS3D_DEFERRED flag and then set it to (2.0, 0.0, 0.0) with the DS3D_IMMEDIATE flag, the velocity will be (2.0, 0.0, 0.0). Then, if your application calls the **CommitDeferredSettings** method, the velocity will not change.

Distance Factor

[This is preliminary documentation and subject to change.]

DirectSound uses meters as the default unit of distance measurements. If your application does not use meters, it can set a distance factor, which is the number of meters in a vector unit.

After you have set the distance factor for a listener, use your application's own distance units in calls to any methods that apply to that listener. Suppose, for example, that the basic unit of measurement in your application is the foot. You call the **IDirectSound3DListener::SetDistanceFactor** method, specifying 0.3048 as the *flDistanceFactor* parameter. (This value is the number of meters in a foot.) From then on, you continue using feet in parameters to method calls, and they are automatically converted to meters.

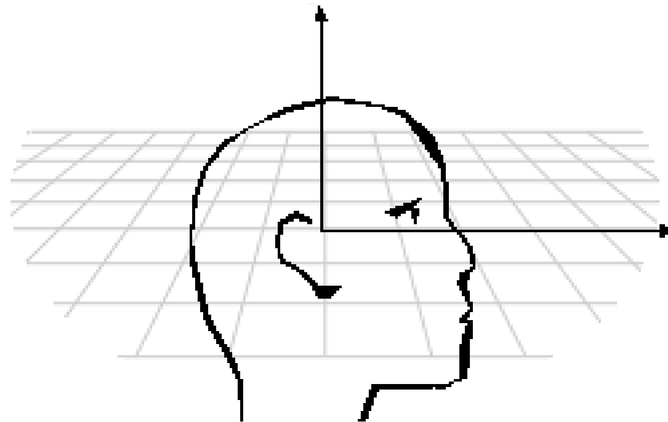
You can retrieve the current distance factor set for a listener by using the **IDirectSound3DListener::GetDistanceFactor** method.

The distance factor mainly affects Doppler shift (by changing the actual velocity represented by *n* units per second). It does not directly affect rolloff, because the rate of attenuation over distance is based on the minimum distance in units. If you set the minimum distance for a given sound at 2 units, the volume will be halved at a distance of 4 units, whether those units are in feet, meters, or any other measure. For more information, see Minimum and Maximum Distances.

Listener Orientation

[This is preliminary documentation and subject to change.]

Listener *orientation* is defined by the relationship between two vectors that share an origin at the center of the listener's head: the *top* and *front vectors*. The top vector points straight up through the top of the head, and the front vector points forward through the listener's face at right angles to the top vector, as in the following illustration.



An application can set and retrieve the listener's orientation by using the **IDirectSound3DListener::SetOrientation** and **IDirectSound3DListener::GetOrientation** methods. By default, the front vector is (0, 0, 1.0), and the top vector is (0, 1.0, 0).

The two vectors must always be at right angles to one another. If necessary, DirectSound will adjust the front vector so that it is at right angles to the top vector.

Listener Position and Velocity

[This is preliminary documentation and subject to change.]

An application can set and retrieve a listener's position in 3-D space by using the **IDirectSound3DListener::SetPosition** and **IDirectSound3DListener::GetPosition** methods.

To set or retrieve the velocity value that DirectSound uses to calculate Doppler-shift effects for a listener, use the **IDirectSound3DListener::SetVelocity** and **IDirectSound3DListener::GetVelocity** methods. Velocity is measured in distance units per second—by default, meters per second.

As with buffers, a listener's position and its velocity are completely independent. It is entirely up to the application to set an appropriate velocity for the listener.

Doppler Factor

[This is preliminary documentation and subject to change.]

DirectSound automatically creates Doppler shift effects for any buffer or listener that has a *velocity*. Effects are cumulative: if the listener and the sound buffer are both moving, the system automatically calculates the relative velocity and adjusts the Doppler effect accordingly.

In order to have realistic Doppler shift effects in your application, you must calculate the speed of any object that is moving and set the appropriate velocity for that sound

source or listener. You are free to exaggerate or lessen this value in a particular case in order to create special effects. You can also globally increase or decrease Doppler shift by changing the Doppler factor.

The Doppler factor can range from DS3D_MINDOPPLERFACTOR to DS3D_MAXDOPPLERFACTOR, currently defined in Dsound.h as 0.0 and 10.0 respectively. A value of 0 means no Doppler shift is applied to a sound. Every other value represents a multiple of the real-world Doppler shift. In other words, a value of 1 (or DS3D_DEFAULTDOPPLERFACTOR) means the Doppler shift that would be experienced in the real world is applied to the sound; a value of 2 means twice the real-world Doppler shift; and so on.

The Doppler factor can be set and retrieved by using the **IDirectSound3DListener::SetDopplerFactor** and **IDirectSound3DListener::GetDopplerFactor** methods.

Rolloff Factor

[This is preliminary documentation and subject to change.]

Rolloff is the amount of attenuation that is applied to sounds, based on the listener's distance from the sound source. DirectSound can ignore rolloff, exaggerate it, or give it the same effect as in the real world, depending on a variable called the rolloff factor.

The rolloff factor can range from DS3D_MINROLLOFFFACTOR to DS3D_MAXROLLOFFFACTOR, currently defined in Dsound.h as 0.0 and 10.0 respectively. A value of DS3D_MINROLLOFFFACTOR means no rolloff is applied to a sound. Every other value represents a multiple of the real-world rolloff. In other words, a value of 1 (DS3D_DEFAULTROLLOFFFACTOR) means the rolloff that would be experienced in the real world is applied to the sound; a value of 2 means twice the real-world rolloff, and so on.

You set and retrieve the rolloff factor by using the **IDirectSound3DListener::SetRolloffFactor** and **IDirectSound3DListener::GetRolloffFactor** methods.

To change the rolloff for an individual sound buffer, you can set the minimum distance for the buffer. For more information, see Minimum and Maximum Distances.

DirectSoundCapture

[This is preliminary documentation and subject to change.]

DirectSoundCapture provides an interface for capturing digital audio data from an input source. To use it you must create an instance of the **IDirectSoundCapture** interface, then use its methods to create a single capture buffer. The actual capturing is done with the methods of the buffer object.

This section covers the following topics:

- Enumeration of Capture Devices
- Creating the DirectSoundCapture Object
- Capture Device Capabilities
- Creating a Capture Buffer
- Capture Buffer Information
- Capture Buffer Notification
- Capturing Sounds

Enumeration of Capture Devices

[This is preliminary documentation and subject to change.]

For an application that is simply going to capture sounds through the user's preferred capture device, you don't need to enumerate the available devices. When you create the DirectSoundCapture object with NULL as the device identifier, the interface will automatically be associated with the default device if one is present. (If no device driver is present, the call to the **DirectSoundCaptureCreate** function fails.)

However, if you are looking for a particular kind of device or wish to offer the user a choice of devices, you must enumerate the devices available on the system.

Enumeration serves three purposes:

- Reports what hardware is available
- Supplies a GUID for each device
- Allows you to initialize DirectSoundCapture for each device as it is enumerated, so that you can check the capabilities of the device

To enumerate devices you must first set up a callback function that will be called each time DirectSound finds a device. You can do anything you want within this function, and you can give it any name, but you must declare it in the same form as **DSEnumCallback**, a prototype in this documentation. The callback function must return TRUE if enumeration is to continue, or FALSE otherwise (for instance, after finding a device with the capabilities you need).

For a sample callback function, see Enumeration of Sound Devices. Note that a GUID for each device is obtained as one of the parameters to this function.

The enumeration is set in motion by using the **DirectSoundCaptureEnumerate** function:

```
DWORD pv;  // Any 32-bit value.

HRESULT hr = DirectSoundCaptureEnumerate(
       (LPDSENUMCALLBACK)DSEnumProc,
       &pv)
```

Creating the DirectSoundCapture Object

[This is preliminary documentation and subject to change.]

You create the DirectSoundCapture object by calling the **DirectSoundCaptureCreate** function, which returns a pointer to an **IDirectSoundCapture** COM interface.

The *lpGUID* parameter to **DirectSoundCaptureCreate** can be a GUID obtained by enumeration, or it can be NULL for the preferred capture device. In most cases you will pass NULL.

You can also use the **CoCreateInstance** function to create the object. The procedure is similar to that for the DirectSound object; see Creating the DirectSound Object. If you use **CoCreateInstance**, then the object is created for the default capture device selected by the user on the multimedia control panel.

If you want DirectSound and DirectSoundCapture objects to coexist, then you should create and initialize the DirectSound object before creating and initializing the DirectSoundCapture object.

Some audio devices aren't configured for full duplex audio by default. If you have problems with creating and initializing both a DirectSound object and a DirectSoundCapture object, you should check your audio device to ensure that two DMA channels are enabled.

Capture Device Capabilities

[This is preliminary documentation and subject to change.]

To retrieve the capabilities of a capture device, call the **IDirectSoundCapture::GetCaps** method. The parameter to this method is a pointer to the **DSCCAPS** structure. As with other such structures, you have to initialize the **dwSize** member before passing it.

On return, the structure contains the number of channels the device supports as well as a combination of values for supported formats, equivalent to the values in the **WAVEINCAPS** structure used in the Win32 waveform audio functions.

Creating a Capture Buffer

[This is preliminary documentation and subject to change.]

You create a capture buffer by calling the **IDirectSoundCapture::CreateCaptureBuffer** method of the DirectSoundCapture object.

One of the parameters to the method is a **DSCBUFFERDESC** structure that describes the characteristics of the desired buffer. The last member of this structure is a **WAVEFORMATEX** structure, which must be initialized with the details of the desired wave format. For more information on this structure, see Sound Data.

Note that if your application is using DirectSound as well as DirectSoundCapture, capture buffer creation can fail when the format of the capture buffer is not the same as that of the primary buffer. The reason is that some cards have only a single clock and cannot support capture and playback at two different frequencies.

The following example sets up a capture buffer that will hold 1 second of data:

```
/* In this example it is assumed that pDSC is a valid pointer to
   a DirectSoundCapture object. */

DSCBUFFERDESC             dscbd;
LPDIRECTSOUNDCAPTUREBUFFER pDSCB;
WAVEFORMATEX              wfx =
    {WAVE_FORMAT_PCM, 2, 44100, 176400, 4, 16, 0};
    // wFormatTag, nChannels, nSamplesPerSec, mAvgBytesPerSec,
    // nBlockAlign, wBitsPerSample, cbSize
dscbd.dwSize = sizeof(DSCBUFFERDESC);
dscbd.dwFlags = 0;
dscbd.dwBufferBytes = wfx.nAvgBytesPerSec;
dscbd.dwReserved = 0;
dscbd.lpwfxFormat = &wfx;

pDSCB = NULL;

HRESULT hr = pDSC->CreateCaptureBuffer(&dscbd,
        &pDSCB, NULL);
```

Capture Buffer Information

[This is preliminary documentation and subject to change.]

Use the **IDirectSoundCaptureBuffer::GetCaps** method to retrieve the size of a capture buffer. Be sure to initialize the **dwSize** member of the **DSCBCAPS** structure before passing it as a parameter.You can also retrieve information about the format of the data in the buffer, as set when the buffer was created. Call the **IDirectSoundCaptureBuffer::GetFormat** method, which returns the format information in a **WAVEFORMATEX** structure. For more information on this structure, see Sound Data.

Note that your application can allow for extra format information in the **WAVEFORMATEX** structure by first calling the **GetFormat** method with NULL as the *lpwfxFormat* parameter. In this case the **DWORD** pointed to by the

lpdwSizeWritten parameter will receive the size of the structure needed to receive complete format information.

To find out what a capture buffer is currently doing, call the **IDirectSoundCaptureBuffer::GetStatus** method. This method fills a **DWORD** variable with a combination of flags that indicate whether the buffer is busy capturing, and if so, whether it is looping; that is, whether the DSCBSTART_LOOPING flag was set in the last call to **IDirectSoundCaptureBuffer::Start**.

Finally, the **IDirectSoundCaptureBuffer::GetCurrentPosition** method returns the current read and capture positions within the buffer. The read position is the end of the data that has been captured into the buffer at this point. The capture position is the end of the block of data that is currently being copied from the hardware. You can safely copy data from the buffer only up to the read position.

Capture Buffer Notification

[This is preliminary documentation and subject to change.]

You may want your application to be notified when the current read position reaches a certain point in the buffer, or when it reaches the end. The current read position is the point up to which it is safe to read data from the buffer. With the **IDirectSoundNotify::SetNotificationPositions** method you can set any number of points within the buffer where events are to be signaled.

First you have to obtain a pointer to the **IDirectSoundNotify** interface. You can do this with the capture buffer's **QueryInterface** method, as shown in the example under Play Buffer Notification.

Next create an event object with the Win32 **CreateEvent** function. You put the handle to this event in the **hEventNotify** member of a **DSBPOSITIONNOTIFY** structure, and in the **dwOffset** member of that structure you specify the offset within the buffer where you want the event to be signaled. Then you pass the address of the structure — or array of structures, if you want to set more than one notification position — to the **IDirectSoundNotify::SetNotificationPositions** method.

The following example sets up three notification positions in a one-second buffer. One event will be signaled when the read position nears the halfway point in the buffer, another will be signaled when it nears the end of the buffer, and the third will be signaled when capture stops.

```
#define cEvents  3

/* In this example it is assumed that the following variables have
   been properly initialized, and that wfx was included in the buffer
   description when the buffer was created.

   LPDIRECTSOUNDNOTIFY  lpDsNotify;
   WAVEFORMATEX         wfx;
```

```
*/

HANDLE             rghEvent[cEvents] = {0};
DSBPOSITIONNOTIFY  rgdsbpn[cEvents];
HRESULT            hr;
int                i;

// Create the events
for (i = 0; i < cEvents; ++i)
{
   rghEvent[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
   if (NULL == rghEvent[i])
   {
      hr = GetLastError();
      goto Error;
   }
}

// Describe notifications.

rgdsbpn[0].dwOffset = (wfx.nAvgBytesPerSec/2) -1;
rgdsbpn[0].hEventNotify = rghEvent[0];

rgdsbpn[1].dwOffset = wfx.nAvgBytesPerSec - 1;
rgdsbpn[1].hEventNotify = rghEvent[1];

rgdsbpn[2].dwOffset = DSBPN_OFFSETSTOP;
rgdsbpn[2].hEventNotify = rghEvent[2];

// Create notifications.

hr = lpDsNotify->SetNotificationPositions(cEvents, rgdsbpn);
```

Capturing Sounds

[This is preliminary documentation and subject to change.]

Capturing a sound consists of the following steps:

1. Start the buffer by calling the **IDirectSoundCaptureBuffer::Start** method. Audio data from the input device begins filling the buffer from the beginning.
2. Wait until the desired amount of data is available. For one method of determining when the capture position reaches a certain point, see Capture Buffer Notification.
3. When sufficient data is available, lock a portion of the capture buffer by calling the **IDirectSoundCaptureBuffer::Lock** method.

To make sure you are not attempting to lock a portion of memory that is about to be used for capture, you can first obtain the current read position by calling **IDirectSoundCaptureBuffer::GetCurrentPosition**. For an explanation of the read position, see Capture Buffer Information.

As parameters to the **Lock** method, you pass the size and offset of the block of memory you want to read. The method returns a pointer to the address where the memory block begins, and the size of the block. If the block wraps around from the end of the buffer to the beginning, two pointers are returned, one for each section of the block. The second pointer is NULL if the locked portion of the buffer does not wrap around.

4. Copy the data from the buffer, using the addresses and block sizes returned by the **Lock** method.
5. Unlock the buffer with the **IDirectSoundCaptureBuffer::Unlock** method.
6. Repeat steps 2 to 5 until you are ready to stop capturing data. Then call the **IDirectSoundCaptureBuffer::Stop** method.

Normally the buffer stops capturing automatically when the capture position reaches the end of the buffer. However, if the DSCBSTART_LOOPING flag was set in the *dwFlags* parameter to the **IDirectSoundCaptureBuffer::Start** method, the capture will continue until the application calls the **IDirectSoundCaptureBuffer::Stop** method.

DirectSound Property Sets

[This is preliminary documentation and subject to change.]

Through the **IKsPropertySet** interface, DirectSound is able to support extended services offered by sound cards and their associated drivers.

Properties are arranged in sets. A **GUID** identifies a set, and a **ULONG** identifies a particular property within the set. For example, a hardware vendor might design a card capable of reverberation effects and define a property set DSPROPSETID_ReverbProps containing properties such as DSPROPERTY_REVERBPROPS_HALL and DSPROPERTY_REVERBPROPS_STADIUM.

Typically, the property identifiers are defined using a C language enumeration starting at ordinal zero.

Individual properties may also have associated parameters. The **IKsPropertySet** interface specification intentionally leaves these parameters undefined, allowing the designer of the property set to use them in a way most beneficial to the properties within the set being designed. The precise meaning of the parameters is defined with the definition of the properties.

To make use of extended properties on sound cards, you must first determine whether the driver supports the **IKsPropertySet** interface, and obtain a pointer to the

interface if it is supported. You can do this by calling the **QueryInterface** method of an existing interface on a DirectSound3DBuffer object.

```
HRESULT hr = lpDirectSound3DBuffer->QueryInterface(
        IID_IKsPropertySet,
        (void**)&lpKsPropertySet));
```

In the example, *lpDirectSound3DBuffer* is a pointer to the buffer's interface and *lpKsPropertySet* receives the address of the **IKsPropertySet** interface if one is found. *IID_IKsPropertySet* is a **GUID** defined in Dsound.h.

The call will succeed only if the buffer is hardware-accelerated and the underlying driver supports property sets. If it does succeed, you can now look for a particular property using the **IKsPropertySet::QuerySupported** method. The value of the *PropertySetId* parameter is a **GUID** defined by the hardware vendor.

Once you've determined that support for a particular property exists, you can change the state of the property by using the **IKsPropertySet::Set** method and determine its present state by using the **IKsPropertySet::Get** method. The state of the property is set or returned in the *pPropertyData* parameter.

Additional property parameters may also be passed to the object in a structure pointed to by the *pPropertyParams* parameter to the **IKsPropertySet::Set** method. The exact way in which this parameter is to be used is defined in the hardware vendor's specifications for the property set, but typically it would be used to define the instance of the property set. In practice, the *pPropertyParams* parameter is rarely used.

Let's take a somewhat whimsical example. Suppose a sound card has the ability to play a set of songs in the voices of several famous tenors. The driver developer creates a property set, DSPROPSETID_Song, containing properties like DSPROPERTY_SONG_IRISH_EYES and DSPROPERTY_SONG_O_SOLE_MIO. The property set applies to all of the tenors, and the driver developer has specified that *pPropertyParams* defines the tenor instance. Now you, the application developer, want to make Caruso sing one of the songs:

```
/* It is assumed that the hardware vendor has also defined
   CARUSO and START in a header file. */

DWORD dwTenor = CARUSO;
BOOL  StartOrStop = START;

HRESULT hr = lpKsPropertySet->Set(
        DSPROPSETID_Song,
        DSPROPERTY_SONG_IRISH_EYES,
        &dwTenor,
        sizeof(dwTenor),
        &StartOrStop,
        sizeof(StartOrStop));
```

Optimizing DirectSound Performance

[This is preliminary documentation and subject to change.]

This section offers some miscellaneous tips for improving the performance of DirectSound. The following topics are covered:

- Matching Buffer Formats
- Playing the Primary Buffer Continuously
- Using Hardware Mixing
- Minimizing Control Changes
- CPU Considerations for 3-D Buffers

Matching Buffer Formats

[This is preliminary documentation and subject to change.]

The DirectSound mixer converts the data from each secondary buffer into the format of the primary buffer. This conversion is done on the fly as data is mixed into the primary buffer, and costs CPU cycles. You can eliminate this overhead by ensuring that your secondary buffers and primary buffer have the same format. Normally this means setting the primary buffer format to the format of the wave files used for data.

Because of the way DirectSound does format conversion, you only need to match the sample rate and number of channels. It doesn't matter if there is a difference in sample size (8-bit or 16-bit).

Playing the Primary Buffer Continuously

[This is preliminary documentation and subject to change.]

When there are no sounds playing, DirectSound stops the mixer engine and halts DMA (direct memory access) activity. If your application has frequent short intervals of silence, the overhead of starting and stopping the mixer each time a sound is played may be worse than the DMA overhead if you kept the mixer active. Also, some sound hardware or drivers may produce unwanted audible artifacts from frequent starting and stopping of playback. If your application is playing audio almost continuously with only short breaks of silence, you can force the mixer engine to remain active by calling the **IDirectSoundBuffer::Play** method for the primary buffer. The mixer will continue to run silently.

To resume the default behavior of stopping the mixer engine when there are no sounds playing, call the **IDirectSoundBuffer::Stop** method for the primary buffer.

For more information, see Access to the Primary Buffer

Using Hardware Mixing

[This is preliminary documentation and subject to change.]

Most sound cards support some level of hardware mixing if there is a DirectSound driver for the card. The following tips will allow you to make the most of hardware mixing:

- Use static buffers for sounds that you want to be mixed in hardware. DirectSound will attempt to use hardware mixing on static buffers.
- Create sound buffers first for the sounds you use the most. There is a limit to the number of buffers that can be mixed by hardware.
- At run time, use the **IDirectSound::GetCaps** method to determine what formats are supported by the sound-accelerator hardware and use only those formats if possible.
- To create a static buffer, specify the DSBCAPS_STATIC flag in the **dwFlags** member of the **DSBUFFERDESC** structure when you create a secondary buffer. You can also specify the DSBCAPS_LOCHARDWARE flag to force hardware mixing for a buffer, however, if you do this and resources for hardware mixing are not available, the **IDirectSound::CreateSoundBuffer** method will fail.

Minimizing Control Changes

[This is preliminary documentation and subject to change.]

Performance is affected when you change the pan, volume, or frequency on a secondary buffer. To prevent interruptions in sound output, the DirectSound mixer must mix ahead from 20 to 100 or more milliseconds. Whenever you make a control change, the mixer has to flush its mix-ahead buffer and remix with the changed sound.

It's a good idea to minimize the number of control changes you send. Try reducing the frequency of calls to routines that use the **IDirectSoundBuffer**::**SetVolume**, **IDirectSoundBuffer::SetPan**, and **IDirectSoundBuffer::SetFrequency** methods. For example, if you have a routine that moves a sound from the left to the right speaker in synchronization with animation frames, try calling the **SetPan** method only every second or third frame.

Note

3-D control changes (orientation, position, velocity, Doppler factor, and so on) also cause DirectSound to remix its mix-ahead buffer. However, you can group a number of 3-D control changes together and cause only a single remix. See Deferred Settings.

CPU Considerations for 3-D Buffers

[This is preliminary documentation and subject to change.]

Software-emulated 3-D buffers are computationally expensive. For example, each buffer can consume about 6 percent of the processing time of a Pentium 90. You should take this into consideration when deciding when and how to use 3-D buffers in your applications.

Use as few 3-D sounds as you can, and don't use 3-D on sounds that won't really benefit from the effect. Design your application so that it's easy to enable and disable 3-D effects on each sound. You can call the **IDirectSound3DBuffer::SetMode** method with the DS3DMODE_DISABLE flag to disable 3-D processing on any 3-D sound buffer.

DirectSound provides a means for hardware manufacturers to provide acceleration of 3-D audio buffers. On these audio cards the host CPU consumption will not be a consideration.

Using Wave Files

[This is preliminary documentation and subject to change.]

The DirectSound and DirectSoundCapture APIs do not include methods for handling wave files. However, the source code for several of the sample applications includes a handy file, Wave.c, which contains functions for opening and creating a wave file, reading and writer the file headers, and streaming data to or from the file.

Wave files are in the Resource Interchange File Format (RIFF), which consists of a variable number of "chunks" containing either header information (for example, the wave format of sound samples) or data (the samples themselves). The Win32 API supplies functions for opening and closing RIFF files, seeking chunks, and so on. These functions, whose names all start with "mmio," are used by the wrapper functions in the Wave.c file.

More information about using the wrapper functions is given in the following two topics:

- Reading from a Wave File
- Writing to a Wave File

Reading from a Wave File

[This is preliminary documentation and subject to change.]

In order to use the wrapper functions in Wave.c, you must declare the following four variables:

```
WAVEFORMATEX  *pwfx;         // Wave format info
HMMIO         hmmio;         // File handle
```

```
MMCKINFO     mmckinfoData;   // Chunk info
MMCKINFO     mmckinfoParent; // Parent chunk info
```

The first step in reading a wave file is to call the **WaveOpenFile** function. This gets a handle to the file, verifies that it is in RIFF format, and gets information about the wave format. The parameters are the filename and the addresses of three of the variables you have declared:

```
if (WaveOpenFile(lpzFileName, &hmmio, &pwfx, &mmckinfoParent) != 0)
{
   // Failure
}
```

Note that the wrapper functions all return zero if successful.

The next step is to call the **WaveStartDataRead** function, causing the file pointer to descend to the data chunk. This function also fills in the **MMCKINFO** structure for the data chunk, so that you know how much data is available:

```
if (WaveStartDataRead(&hmmio, &mmckinfoData, &mmckinfoParent) != 0)
   {
   // Failure
   }
```

The application can now begin copying data from the file to a secondary sound buffer. Normally you don't create the sound buffer until you have obtained the size of the data chunk and the format of the wave. The following code creates a static buffer just large enough to hold all the data in the file.

```
/* It is assumed that lpds in a valid pointer
   to the DirectSound object. */

LPDIRECTSOUNDBUFFER  lpdsbStatic;
DSBUFFERDESC         dsbdesc;

memset(&dsbdesc, 0, sizeof(DSBUFFERDESC));
dsbdesc.dwSize = sizeof(DSBUFFERDESC);
dsbdesc.dwFlags = DSBCAPS_STATIC;
dsbdesc.dwBufferBytes = mmckinfoData.cksize;
dsbdesc.lpwfxFormat = pwfx;

if FAILED(lpds->CreateSoundBuffer(&dsbdesc, &lpdsbStatic, NULL))
{
   WaveCloseReadFile(&hmmio, &pwfx);
   return FALSE;
}
```

Because in this case the application is not streaming the data but simply filling a static buffer, the entire buffer is locked from the beginning. There is no wraparound, so only a single pointer and byte count are required.

```
LPVOID lpvAudio1;
DWORD  dwBytes1;

if FAILED(lpdsbStatic->Lock(
        0,              // Offset of lock start
        0,              // Size of lock; ignored in this case
        &lpvAudio1,     // Address of lock start
        &dwBytes1,      // Number of bytes locked
        NULL,           // Wraparound start; not used
        NULL,           // Wraparound size; not used
        DSBLOCK_ENTIREBUFFER))  // Flag
{
    // Error handling
    WaveCloseReadFile(&hmmio, &pwfx);
    .
    .
    .
}
```

The **WaveReadFile** function in Wave.c copies the data from the file to the buffer pointer and returns zero if successful.

```
UINT cbBytesRead;

if (WaveReadFile(
        hmmio,              // file handle
        dwBytes1,           // no. of bytes to read
        (BYTE *) lpvAudio1, // destination
        &mmckinfoData,      // file chunk info
        &cbBytesRead))      // actual no. of bytes read
{
    // Handle failure on non-zero return
    WaveCloseReadFile(&hmmio, &pwfx);
    .
    .
    .
}
```

Finally, the application unlocks the buffer and closes the wave file:

```
lpdsbStatic->Unlock(lpvAudio1, dwBytes1, NULL, 0);
WaveCloseReadFile(&hmmio, &pwfx);
```

For a streaming buffer, you would typically call **WaveReadFile** at regular intervals determined by the current play position. (See Play Buffer Notification.) If the locked portion of the buffer wrapped around, of course, you would call **WaveReadFile** once for each segment of the lock.

Writing to a Wave File

[This is preliminary documentation and subject to change.]

To prepare for writing to a wave file, the application must first declare four variables to be passed to the functions in Wave.c:

```
WAVEFORMATEX  wfx;           // Wave format info
HMMIO         hmmio;         // File handle
MMCKINFO      mmckinfoData;   // Chunk info
MMCKINFO      mmckinfoParent; // Parent chunk (file) info
```

You must also initialize the **WAVEFORMATEX** structure with the format of the capture buffer.

Now you call the **WaveCreateFile** function, passing in the desired filename and the addresses of the global variables. The function creates the file and writes some header information. Like other functions in Wave.c, **WaveCreateFile** returns zero if successful.

```
if (WaveCreateFile(pszFileName, &hmmio, &wfx,
      &mmckinfoData, &mmckinfoParent))
{
   // Failure
}
```

Next, call the **WaveStartDataWrite** function, which initializes the data chunk.

```
if (WaveStartDataWrite(&hmmio, &mmckinfoData, &mmioinfo))
{
    // Failure
}
```

The file is now ready to receive data. The following fragment illustrates how data might be copied from a capture buffer to a file.

```
/* It is assumed that the following variables contain
   valid assignments:
LPDIRECTSOUNDCAPTUREBUFFER lpdscb;    // Capture buffer
DSCBUFFERDESC    dscbDesc;            // Capture buffer description
DWORD            dwReadCursor;        // Internal cursor in buffer
DWORD            dwNumBytes;          // Bytes available
DWORD            dwTotalBytesWritten; // Running total in file
*/
```

```
LPBYTE pbInput1, pbInput2; // Pointers to data in buffer
DWORD  cbInput1, cbInput2; // Count of bytes in locked portion
UINT   BytesWritten;       // Count of bytes written to file

if FAILED(hr = lpdscb->Lock(dwReadCursor, dwNumBytes,
                  (LPVOID *)&pbInput1, &cbInput1,
                  (LPVOID *)&pbInput2, &cbInput2, 0))
{
   // Failure
}
else
{
   if (WaveWriteFile(hmmio, cbInput1, pbInput1, &mmckinfoData,
                &dwBytesWritten, &mmioinfo))
   {
      // Failure
   }
   else dwTotalBytesWritten += BytesWritten;
   if (pbInput2 != NULL)
   {
      if (WaveWriteFile(hmmio, cbInput2, pbInput2, &mmckinfoData,
                &BytesWritten, &mmioinfo))
      {
         // Failure
      }
      else dwTotalBytesWritten += BytesWritten;

   }
   lpdscb->Unlock(pbInput1, cbInput1, pbInput2, cbInput2);

   // Increment internal cursor, compensating for wrap around
   dwReadCursor += dwNumBytes;
   while (dwReadCursor >= dscbDesc.dwBufferBytes)
            dwMyReadCursor -= dscbDesc.dwBufferBytes;

}
```

When you are finished capturing data, you close the file:

```
WaveCloseWriteFile(&hmmio, &mmckinfoData,
      &mmckinfoParent, &mmioinfo,
      dwTotalBytesWritten / (wfx.wBitsPerSample / 8));
```

The **WaveCloseWriteFile** function calculates the total number of samples in the file and writes this number to the data chunk header.

Reading Wave Data from a Resource

[This is preliminary documentation and subject to change.]

The DirectSound API does not include methods for reading a wave from a resource. However, there are helper functions in Dsutil.c, which you will find in the Sdk\Samples\Misc folder.

To store wave sounds in an executable or DLL, import your wave files as resources and give them string names. Note that Dsutil.c expects these resources to be of type "WAV." If you are using Microsoft® Visual C++®, imported wave files are turned into resources of type "WAVE." You must either change this nomenclature to "WAV" by editing the resource file, or else modify the assignment of *c_szWAV* in Dsutil.c so that the **FindResource** function is looking for "WAVE" resources.

You may also want to modify Dsutil.c so that the **DSLoadSoundBuffer** function sets only the appropriate control flags.

To create a static buffer and load the sound into it, simply pass the **IDirectSound** interface pointer and the name of the resource to the **DSLoadSoundBuffer** function. If successful, the function returns a pointer to the buffer. Here's a sample call, where *lpds* is the pointer to the **IDirectSound** interface:

```
#include "dsutil.h"

LPDIRECTSOUNDBUFFER  lpdsbFootstep;

lpdsbFootstep = DSLoadSoundBuffer(lpds, "FOOTSTEP");
if (lpdsbFootstep == NULL)
{
   // Failure
}
```

DirectSound Tutorials

[This is preliminary documentation and subject to change.]

This section contains tutorials that provide step-by-step instructions for implementing basic DirectSound functionality.

- Tutorial 1: Sound Playback

  The first tutorial shows how to set up the DirectSound system, create a secondary buffer, and play data from a wave file.
- Tutorial 2: Sound Capture

  The second tutorial shows how to create a DirectSoundCapture object and a capture buffer, and how to write data to a wave file.

Tutorial 1: Sound Playback

[This is preliminary documentation and subject to change.]

This tutorial shows how to create a simple DirectSound application that will play a wave file of any size.

The functions for opening, reading, and closing wave files are in Wave.c, a module that is found with the DSShow3D sample application in the DirectX SDK. In order to implement the techniques shown in the tutorial, you must add Wave.c and Wave.h to your project and link to Winmm.lib. You must also add Debug.c and Debug.h from the same sample directory, or else edit the calls to the **ASSERT** macro in Wave.c to call the standard **assert** function.

The method calls in this tutorial are made through the macros defined in Dsound.h, which are valid for both C and C++.

The tutorial is broken down into the following steps:

- Step 1: Setting Up DirectSound
- Step 2: Opening the Wave File
- Step 3: Creating the Secondary Buffer
- Step 4: Setting Up Play Notification
- Step 5: Handling the Play Notifications
- Step 6: Streaming Data from the Wave File
- Step 7: Shutting Down DirectSound

Step 1: Setting Up DirectSound

[This is preliminary documentation and subject to change.]

The tutorial requires the following definitions and global variables:

```
#define NUMEVENTS 2

LPDIRECTSOUND            lpds;
DSBUFFERDESC             dsbdesc;
LPDIRECTSOUNDBUFFER      lpdsb;
LPDIRECTSOUNDBUFFER      lpdsbPrimary;
LPDIRECTSOUNDNOTIFY      lpdsNotify;
WAVEFORMATEX             *pwfx;
HMMIO                    hmmio;
MMCKINFO                 mmckinfoData, mmckinfoParent;
DSBPOSITIONNOTIFY        rgdsbpn[NUMEVENTS];
HANDLE                   rghEvent[NUMEVENTS];
```

The first step is to create the DirectSound object, establish a cooperative level, and set the primary buffer format. All this is done in the InitDSound function shown in the following code.

The function takes two parameters: the main window handle and a pointer to the GUID of the sound device. In most cases you will pass NULL as the second parameter, indicating the default device, but you may obtain a GUID by device enumeration.

```
BOOL InitDSound(HWND hwnd, GUID *pguid)
{
  // Create DirectSound

   if FAILED(DirectSoundCreate(pguid, &lpds, NULL))
      return FALSE;

   // Set co-op level

   if FAILED(IDirectSound_SetCooperativeLevel(
         lpds, hwnd, DSSCL_PRIORITY))
      return FALSE;
```

You have set the priority cooperative level in order to be able to set the format of the primary buffer. If you don't change the default format, output will be in the 8-bit, 22 kHz format regardless of the format of the input. Setting the primary buffer to a higher format can do no harm, because even if the secondary buffers are in a lower format, the samples will be converted automatically by DirectSound. Note also that there's no danger of the call to **IDirectSoundBuffer::SetFormat** failing because the hardware doesn't support the higher format. DirectSound will simply set the closest available format.

To set the format of the primary buffer, you first describe it in the global **DSBUFFERDESC** structure, then pass this description to the **IDirectSound::CreateSoundBuffer** method.

```
   // Obtain primary buffer

   ZeroMemory(&dsbdesc, sizeof(DSBUFFERDESC));
   dsbdesc.dwSize = sizeof(DSBUFFERDESC);
   dsbdesc.dwFlags = DSBCAPS_PRIMARYBUFFER;
   if FAILED(lpds->CreateSoundBuffer(&dsbdesc, &lpdsbPrimary, NULL))
      return FALSE;
```

Once you have a primary buffer object, you describe the desired wave format and pass the description to the **IDirectSoundBuffer::SetFormat** method:

```
   // Set primary buffer format

   WAVEFORMATEX wfx;
```

```
    memset(&wfx, 0, sizeof(WAVEFORMATEX));
    wfx.wFormatTag = WAVE_FORMAT_PCM;
    wfx.nChannels = 2;
    wfx.nSamplesPerSec = 44100;
    wfx.wBitsPerSample = 16;
    wfx.nBlockAlign = wfx.wBitsPerSample / 8 * wfx.nChannels;
    wfx.nAvgBytesPerSec = wfx.nSamplesPerSec * wfx.nBlockAlign;

    hr = lpdsbPrimary->SetFormat(&wfx);

    return TRUE;
} // InitDSound()
```

Step 2: Opening the Wave File

[This is preliminary documentation and subject to change.]

The sample program is a general-purpose wave-file reader that does not expect sound data in a particular format. That being the case, it's necessary to put off creating the secondary sound buffer until the format of the data is known.

The steps in opening the file and creating the secondary buffer are grouped in the SetupStreamBuffer function. This function first performs any necessary cleanup from the last time a sound was played:

```
BOOL SetupStreamBuffer(LPSTR lpzFileName)
{
    // Close any open file and release interfaces

    WaveCloseReadFile(&hmmio, &pwfx);  // Function in Wave.c

    if (lpdsNotify != NULL)
    {
        lpdsNotify->Release();
        lpdsNotify = NULL;
    }

    if (lpdsb != NULL)
    {
        lpdsb->Release();
        lpdsb = NULL;
    }
```

The function now opens a wave file, gets the format, and advances the file pointer to the beginning of the sound data. It does this by using two functions in Wave.c.

```
    if (WaveOpenFile(lpzFileName, &hmmio, &pwfx,
```

```
        &mmckinfoParent) != 0)
    return FALSE;
if (WaveStartDataRead(&hmmio, &mmckinfoData,
        &mmckinfoParent) != 0)
    return FALSE;
```

The WaveOpenFile function initializes three of the global variables declared at the beginning of the tutorial: a file handle, a pointer to a **WAVEFORMATEX** structure, and the **MMCKINFO** structure for the parent chunk (that is, information about the file as a whole).

The file handle and chunk information are then passed to the WaveStartDataRead function, which returns information about the data chunk in *mmckinfoData*. You would be interested in this structure if you were creating a static buffer just big enough to accommodate all the data bytes. In this tutorial, though, you're creating a streaming buffer, so you don't need to know the size of the data chunk.

Step 3: Creating the Secondary Buffer

[This is preliminary documentation and subject to change.]

Still inside the SetupStreamBuffer function, you now create a secondary sound buffer in the same format as the wave file. The process is similar for that you used in Step 1 to create a primary buffer. First you describe the buffer in the global **DSBUFFERDESC** structure, then you pass this description to the **IDirectSound::CreateSoundBuffer** method.

```
memset(&dsbdesc, 0, sizeof(DSBUFFERDESC));
dsbdesc.dwSize = sizeof(DSBUFFERDESC);
dsbdesc.dwFlags =
        DSBCAPS_GETCURRENTPOSITION2   // Always a good idea
        | DSBCAPS_GLOBALFOCUS         // Allows background playing
        | DSBCAPS_CTRLPOSITIONNOTIFY; // Needed for notification

// The size of the buffer is arbitrary, but should be at least
// two seconds, to keep data writes well ahead of the play
// position.

dsbdesc.dwBufferBytes = pwfx->nAvgBytesPerSec * 2;
dsbdesc.lpwfxFormat = pwfx;

if FAILED(IDirectSound_CreateSoundBuffer(
        lpds, &dsbdesc, &lpdsb, NULL))
{
    WaveCloseReadFile(&hmmio, &pwfx);
    return FALSE;
}
```

Step 4: Setting Up Play Notification

[This is preliminary documentation and subject to change.]

Now that you've successfully created a streaming buffer, you ask to be notified whenever the current play position reaches certain points in the buffer, so you'll know when it's time to stream more data. In the example, those positions will be set at the beginning and halfway mark of the buffer.

First you create the required number of events and store their handles in the *rghEvent* array:

```
for (int i = 0; i < NUMEVENTS; i++)
{
   rghEvent[i] = CreateEvent(NULL, FALSE, FALSE, NULL);
   if (NULL == rghEvent[i]) return FALSE;
}
```

Next, you initialize the array of **DSBPOSITIONNOTIFY** structures, each of which associates a position in the buffer with an event handle:

```
rgdsbpn[0].dwOffset = 0;
rgdsbpn[0].hEventNotify = rghEvent[0];
rgdsbpn[1].dwOffset = (dsbdesc.dwBufferBytes/2);
rgdsbpn[1].hEventNotify = rghEvent[1];
```

Then you get the **IDirectSoundNotify** interface from the secondary buffer and pass the **DSBPOSITIONNOTIFY** array to the **SetNotificationPositions** method:

```
if FAILED(IDirectSoundBuffer_QueryInterface(lpdsb,
       IID_IDirectSoundNotify, (VOID **)&lpdsNotify))
   return FALSE;

if FAILED(IDirectSoundNotify_SetNotificationPositions(
        lpdsNotify, NUMEVENTS, rgdsbpn))
{
   IDirectSoundNotify_Release(lpdsNotify);
   return FALSE;
}
```

That completes the setup of the streaming buffer. Because you've already opened the wave file and are ready to start streaming data, you can set the buffer in motion here. You want it to continue running till the complete sound has been played, so you set the DSBPLAY_LOOPING flag.

```
IDirectSoundBuffer_Play(lpdsb, 0, 0, DSBPLAY_LOOPING);
```

```
    return TRUE;
} // end of SetupStreamBuffer()
```

Step 5: Handling the Play Notifications

[This is preliminary documentation and subject to change.]

Notifications are received in the form of signaled events in the message loop within the **WinMain** function. The following fragment illustrates how the loop might be written in order to intercept signaled events as well as standard messages:

```
BOOL Done = FALSE;
while (!Done)
{
    DWORD dwEvt = MsgWaitForMultipleObjects(
            NUMEVENTS,      // How many possible events
            rghEvent,       // Location of handles
            FALSE,          // Wait for all?
            INFINITE,       // How long to wait
            QS_ALLINPUT);   // Any message is an event

    // WAIT_OBJECT_0 == 0 but is properly treated as an arbitrary
    // index value assigned to the first event, therefore we subtract
    // it from dwEvt to get the zero-based index of the event.

    dwEvt -= WAIT_OBJECT_0;

    // If the event was set by the buffer, there's input
    // to process.

    if (dwEvt < NUMEVENTS)
    {
        StreamToBuffer(dwEvt); // copy data to output stream
    }

    // If it's the last event, it's a message

    else if (dwEvt == NUMEVENTS)
    {
        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            if (msg.message == WM_QUIT)
            {
                Done = TRUE;
            }
            else
```

```
        {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
        }
      }
    }  // end message processing
} // while (!Done)
```

Step 6: Streaming Data from the Wave File

[This is preliminary documentation and subject to change.]

The purpose of the notifications you set up and handled in the previous two steps is to alert you whenever half the data in the buffer has been played. As soon as the current play position passes the beginning or halfway point of the buffer, you write data into the segment of the buffer that has just been played. (When the buffer starts, you immediately receive a notification and write to the second half of the buffer.) Because the buffer holds 2 seconds' worth of data, your data write begins almost 1 second ahead of the current play position, which should allow ample time for the process to be completed before the play position reaches the new data.

In the **WinMain** function, whenever an event was signaled you passed the index of that event to the StreamToBuffer function. This index corresponds to the index of the notification position in the **DSBPOSITIONNOTIFY** array.

Here is the first part of the StreamToBuffer function:

```
BOOL StreamToBuffer(DWORD dwPos)
{
  LONG          lNumToWrite;
  DWORD         dwStartOfs;
  VOID          *lpvPtr1, *lpvPtr2;
  DWORD         dwBytes1, dwBytes2;
  UINT          cbBytesRead;
  static DWORD  dwStopNextTime = 0xFFFF;

  if (dwStopNextTime == dwPos)   // All data has been played
  {
    lpdsb->Stop();
    dwStopNextTime = 0xFFFF;
    return TRUE;
  }

  if (dwStopNextTime != 0xFFFF)  // No more to stream, but keep
                                 // playing to end of data
    return TRUE;
```

The *dwStopNextTime* variable is a flag to indicate that the end of the file has been reached; this is set later in the function. If it is nonzero, there is no more data to be streamed. If the value of *dwStopNextTime* equals the index of the notification being handled, then you know that the current play position has returned to where it was when the end of the file was reached; that is, the last segment of data you copied to the buffer has been played. In this case, it's time to stop the buffer so it doesn't keep playing old data.

The next part of the StreamToBuffer function determines the offset within the buffer where you will start copying the new data. Although the buffer in this tutorial has only two notification positions, the code is designed to work with any number. Remember, *dwPos* is the index of the notification position that has just been passed by the current play position. You are going to write to the segment of the buffer that starts at the previous notification position.

```
if (dwPos == 0)
   dwStartOfs = rgdsbpn[NUMEVENTS - 1].dwOffset;
else
   dwStartOfs = rgdsbpn[dwPos-1].dwOffset;
```

Now you determine the size of this segment of the buffer:

```
lNumToWrite = (LONG) rgdsbpn[dwPos].dwOffset - dwStartOfs;
if (lNumToWrite < 0) lNumToWrite += dsbdesc.dwBufferBytes;
```

You now have all the information you need to lock the buffer in preparation for the data write.

```
IDirectSoundBuffer_Lock(lpdsb,
        dwStartOfs,      // Offset of lock start
        lNumToWrite,     // Number of bytes to lock
        &lpvPtr1,        // Address of lock start
        &dwBytes1,       // Count of bytes locked
        &lpvPtr2,        // Address of wrap around
        &dwBytes2,       // Count of wrap around bytes
        0);              // Flags
```

In this example the lock will never wrap around, so you need to do only a single data copy from the file. You get the data by calling the WaveReadFile function in Wave.c:

```
WaveReadFile(hmmio,              // File handle
      dwBytes1,           // Number of bytes to get
      (BYTE *) lpvPtr1,   // Destination
      &mmckinfoData,      // File chunk info
      &cbBytesRead);      // Actual bytes read
```

Now determine if the end of the file has been reached. If it has, close it, write silence to the rest of this segment (because it will play all the way through before the buffer is stopped), and set the *dwStopNextTime* flag.

```
  if (cbBytesRead < dwBytes1)       // Reached end of file
  {
    WaveCloseReadFile(&hmmio, &pwfx);
    FillMemory((PBYTE)lpvPtr1 + cbBytesRead,
          dwBytes1 - cbBytesRead,
          (dsbdesc.lpwfxFormat->wBitsPerSample==8) ? 128 : 0);
    dwStopNextTime = dwPos;
  }
```

Finally, unlock the buffer and return to the message loop:

```
  IDirectSoundBuffer_Unlock(lpdsb,
        lpvPtr1, dwBytes1, lpvPtr2, dwBytes2);

  return TRUE;
} // end StreamToBuffer()
```

Step 7: Shutting Down DirectSound

[This is preliminary documentation and subject to change.]

Before closing, your application needs to close any open wave file and release the DirectSoundNotify and DirectSound objects. Note that because you queried for the **IDirectSoundNotify** interface from a DirectSoundBuffer object, you must release the DirectSoundNotify object before releasing DirectSound. Releasing DirectSound automatically releases any existing buffers.

The following function does all the necessary cleanup:

```
void DSExit(void)
{
  WaveCloseReadFile(&hmmio, &pwfx);
  if (lpdsNotify)
    lpdsNotify->Release();
  if (lpds)
    lpds->Release();
}
```

Tutorial 2: Sound Capture

[This is preliminary documentation and subject to change.]

This tutorial shows how to implement DirectSoundCapture to capture a sound (typically from the microphone input) and write it to a wave file.

The functions for creating, writing, and closing wave files are in Wave.c, a module found with the DSShow3D sample application in the DirectX SDK. In order to implement the techniques shown in the tutorial, you must add Wave.c and Wave.h to your project and link to Winmm.lib. You must also add Debug.c and Debug.h from the same sample directory, or else edit the calls to the **ASSERT** macro in Wave.c to call the standard **assert** function.

The method calls in this tutorial are made through the macros defined in Dsound.h, which are valid for both C and C++.

The tutorial is broken down into the following steps:

- Step 1: Setting Up DirectSoundCapture
- Step 2: Setting the Capture Format
- Step 3: Creating the Capture Buffer
- Step 4: Setting Up Capture Notification
- Step 5: Creating the Wave File
- Step 6: Handling the Capture Notifications
- Step 7: Streaming Data to the Wave File
- Step 8: Stopping Capture
- Step 9: Shutting Down DirectSoundCapture

Step 1: Setting Up DirectSoundCapture

[This is preliminary documentation and subject to change.]

The tutorial requires the following definitions and global declarations:

```
#define NUMCAPTUREEVENTS 2

LPDIRECTSOUNDCAPTURE       lpdsc;
LPDIRECTSOUNDCAPTUREBUFFER lpdscb;
LPDIRECTSOUNDNOTIFY        lpdsNotify;
DSCBUFFERDESC              dscbDesc;
HANDLE                     rghEvent[NUMCAPTUREEVENTS];
DSBPOSITIONNOTIFY          rgdscbpn[NUMCAPTUREEVENTS];
WAVEFORMATEX               wfx =
             {WAVE_FORMAT_PCM, 1, 22050, 44100, 2, 16, 0};
HMMIO                      hmmio;
MMCKINFO                   mmckinfoData, mmckinfoParent;
MMIOINFO                   mmioinfo;
DWORD                      dwTotalBytesWritten;
```

It is not necessary to create a DirectSound object in order to use DirectSoundCapture. However, if your application will be playing back sound as well as recording it, you should create DirectSound first.

In this tutorial, all the initialization of the capture system takes place in a function called InitDSoundCapture, which takes no parameters. The first step is to create the DirectSoundCapture object. You associate the object with the default capture device by passing NULL as the first parameter to the **DirectSoundCaptureCreate** function.

```
BOOL InitDSoundCapture(void)
{

  if FAILED(DirectSoundCaptureCreate(NULL, &lpdsc, NULL))
    return FALSE;
```

Step 2: Setting the Capture Format

[This is preliminary documentation and subject to change.]

After creating DirectSoundCapture, you should choose a wave format that is supported by the user's device. One way to do this is to start with the highest format and attempt to create a capture buffer in each format until the creation method succeeds. (For an example of how to step through the various formats, see Fdaudio.cpp in the Fdfilter sample program.) In this tutorial you simply check the capabilities of the device to see whether the default 16-bit format (defined in the declaration of the **WAVEFORMATEX** structure) is supported; if it is not, you select an 8-bit format that is supported by all devices.

The following code is within the InitDSoundCapture function:

```
  DSCCAPS dsccaps;

  dsccaps.dwSize = sizeof(DSCCAPS);
  if FAILED(IDirectSoundCapture_GetCaps(lpdsc, &dsccaps))
    return FALSE;

  if ((dsccaps.dwFormats & WAVE_FORMAT_2M16) == 0)
  {
    wfx.nSamplesPerSec = 11025;
    wfx.nAvgBytesPerSec = 11025;
    wfx.nBlockAlign = 1;
    wfx.wBitsPerSample = 8;
  }
```

Remember that if you are also using DirectSound playback in your application, you must select a capture format that is compatible with the format of the primary buffer. For more information, see Creating a Capture Buffer.

Step 3: Creating the Capture Buffer

[This is preliminary documentation and subject to change.]

Once you have ensured that the **WAVEFORMATEX** structure is valid for the user's device, you can go ahead and create a capture buffer in that format.

```
dscbDesc.dwSize = sizeof(DSCBUFFERDESC);
dscbDesc.dwFlags = 0;
// Buffer will hold one second's worth of audio
dscbDesc.dwBufferBytes = wfx.nAvgBytesPerSec;
dscbDesc.dwReserved = 0;
dscbDesc.lpwfxFormat = &wfx;

if FAILED(IDirectSoundCapture_CreateCaptureBuffer(lpdsc,
      &dscbDesc, &lpdscb, NULL))
   return FALSE;
```

You now have a pointer to the buffer object in *lpdscb*.

Step 4: Setting Up Capture Notification

[This is preliminary documentation and subject to change.]

As the final initialization step in the InitDSoundCapture function, you will set up notification positions in the capture buffer so that the application knows when it's time to stream more data to the file. In the example, these positions are set at the beginning and halfway mark of the buffer.

First you create the required number of events and store their handles in the *rghEvent* array:

```
for (int i = 0; i < NUMCAPTUREEVENTS; i++)
{
   rghEvent[i] = CreateEvent(NULL, FALSE, FALSE, NULL);
   if (NULL == rghEvent[i]) return FALSE;
}
```

Next, you initialize the array of **DSBPOSITIONNOTIFY** structures, each of which associates a position in the buffer with an event handle:

```
rgdscbpn[0].dwOffset = 0;
rgdscbpn[0].hEventNotify = rghEvent[0];
rgdscbpn[1].dwOffset = dscbDesc.dwBufferBytes/2;
rgdscbpn[1].hEventNotify = rghEvent[1];
```

Finally, you get the **IDirectSoundNotify** interface from the capture buffer and pass the **DSBPOSITIONNOTIFY** array to the **SetNotificationPositions** method:

```
    if FAILED(IDirectSoundCaptureBuffer_QueryInterface(lpdscb,
            IID_IDirectSoundNotify, (VOID **)&lpdsNotify))
        return FALSE;

    if FAILED(IDirectSoundNotify_SetNotificationPositions(lpdsNotify,
            NUMCAPTUREEVENTS, rgdscbpn))
    {
        IDirectSoundNotify_Release(lpdsNotify);
        return FALSE;
    }

    return TRUE;
} // end InitDSoundCapture()
```

Note that if you need to set up notifications for both a capture buffer and a secondary (output) buffer, the event handles have to be stored in the same *rghEvent* array. When you receive the notifications in the message loop, you can distinguish between the two types of events by the index number.

Step 5: Creating the Wave File

[This is preliminary documentation and subject to change.]

At this point it is presumed that you have obtained a valid filename and are ready to start saving sound data in a wave file. The process is initiated in the following function:

```
BOOL StartWrite(TCHAR *pszFileName)
{
    if (WaveCreateFile(pszFileName, &hmmio, &wfx,
            &mmckinfoData, &mmckinfoParent))
      return FALSE;

    if (WaveStartDataWrite(&hmmio, &mmckinfoData, &mmioinfo))
    {
        WaveCloseWriteFile(&hmmio, &mmckinfoData,
                &mmckinfoParent, &mmioinfo,
                dwTotalBytesWritten / (wfx.wBitsPerSample / 8));
        DeleteFile(pszFileName);
        return FALSE;
    }
    if FAILED(IDirectSoundCaptureBuffer_Start(lpdscb,
            DSCBSTART_LOOPING))
    {
        WaveCloseWriteFile(&hmmio, &mmckinfoData,
                &mmckinfoParent, &mmioinfo, 0);
        DeleteFile(pszFileName);
```

```
        return FALSE;
    }

    dwTotalBytesWritten = 0;

    return TRUE;
}
```

This function first calls the WaveCreateFile function in Wave.c, in order to create a RIFF file and write the header for the wave format. It then calls the WaveStartDataWrite function, which advances the file pointer to the data chunk. Finally, it starts the capture buffer. In half a second your application will be notified that data is available, and you must be ready to copy it to the file.

Note also the initialization of *dwTotalBytesWritten*. This value is going to be needed for the data chunk header after capture is complete.

Step 6: Handling the Capture Notifications

[This is preliminary documentation and subject to change.]

You receive capture notifications as events in the message loop, just as with playback notifications. Here is a sample loop:

```
BOOL Done = FALSE;
while (!Done)
{
    DWORD dwEvt = MsgWaitForMultipleObjects(
            NUMCAPTUREEVENTS,  // How many possible events
            rghEvent,          // Location of handles
            FALSE,             // Wait for all?
            INFINITE,          // How long to wait
            QS_ALLINPUT);      // Any message is an event

    dwEvt -= WAIT_OBJECT_0;

    // If the event was set by the buffer, there's input
    // to process.

    if (dwEvt < NUMCAPTUREEVENTS)
    {
        StreamToFile();
    }

    // If it's the last event, it's a message
```

```
    else if (dwEvt == NUMCAPTUREEVENTS)
    {
      while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
      {
        if (msg.message == WM_QUIT)
        {
          Done = TRUE;
        }
        else
        {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
        }
      }
    }  // end message processing
  } // while (!Done)
```

As already noted, if you are receiving notifications from both a capture buffer and a secondary (output) buffer, you need to distinguish between the two types of events by the index value in *dwEvt*. For example, events 0 and 1 might be playback notifications, and events 2 and 3 might be capture notifications.

Step 7: Streaming Data to the Wave File

[This is preliminary documentation and subject to change.]

In the first tutorial you saw how to identify the segment of the buffer that was safe to write to by checking the offset of the notification position. This technique would work equally well with the capture buffer in the present tutorial, but this time you'll do things a bit differently.

In the previous step you saw how the StreamToFile function was called in response to a notification event. Unlike the StreamToBuffer function in the previous tutorial, this function does not take the event index as a parameter.

```
BOOL StreamToFile(void)
{
  DWORD          dwReadPos;
  DWORD          dwNumBytes;
  LPBYTE         pbInput1, pbInput2;
  DWORD          cbInput1, cbInput2;
  static DWORD   dwMyReadCursor = 0;
  UINT           dwBytesWritten;
```

Note the static declaration of *dwMyReadCursor*. This is the offset of the next byte of data you want to read; in other words, the byte just beyond the last one read on the previous pass through this function.

The first thing the function does is find the current read position. Remember, this position marks the leading edge of the data that is safe to read. It is not necessarily the same as the notification position, because it has likely advanced since the event was signaled.

```
IDirectSoundCaptureBuffer_GetCurrentPosition(lpdscb,
      NULL, &dwReadPos);
```

The function then subtracts your internal read cursor from the current read position (after allowing for wraparound) in order to determine how many bytes of new data are available:

```
if (dwReadPos < dwMyReadCursor)
      dwReadPos += dscbDesc.dwBufferBytes;
dwNumBytes = dwReadPos - dwMyReadCursor;
```

You then lock the buffer and do the copy. Because the segment of data you've identified as available is not exactly demarcated by the notification positions at the beginning and midpoint of the buffer, the locked portion of the buffer might wrap around, in which case two separate copy operations are required:

```
if FAILED(IDirectSoundCaptureBuffer_Lock(lpdscb,
      dwMyReadCursor, dwNumBytes,
      (LPVOID *)&pbInput1, &cbInput1,
      (LPVOID *)&pbInput2, &cbInput2, 0))
   OutputDebugString("Capture lock failure");

else
   {
   if (WaveWriteFile(hmmio, cbInput1, pbInput1, &mmckinfoData,
            &dwBytesWritten, &mmioinfo))
     OutputDebugString("Failure writing data to file\n");
   dwTotalBytesWritten += dwBytesWritten;

   // Wraparound
   if (pbInput2 != NULL)
   {
      if (WaveWriteFile(hmmio, cbInput2, pbInput2,
            &mmckinfoData, &dwBytesWritten, &mmioinfo))
        OutputDebugString("Failure writing data to file\n");
      dwTotalBytesWritten += dwBytesWritten;
   }

   IDirectSoundCaptureBuffer_Unlock(lpdscb,
```

```
            pbInput1, cbInput1,
            pbInput2, cbInput2);
    }
```

The WaveWriteFile function returns 0 if successful and also fills *dwBytesWritten* with the number of bytes actually copied to the file. This value is added to the cumulative total, which will be needed when the file is closed.

Finally, update the internal read cursor, compensating for wraparound, and return to the message loop:

```
    dwMyReadCursor += dwNumBytes;
    if (dwMyReadCursor >= dscbDesc.dwBufferBytes)
            dwMyReadCursor -= dscbDesc.dwBufferBytes;

    return TRUE;
} // end StreamToFile()
```

Step 8: Stopping Capture

[This is preliminary documentation and subject to change.]

When it's time to stop recording, call the following function:

```
BOOL StopWrite()
{
    IDirectSoundCaptureBuffer_Stop(lpdscb);
    StreamToFile();
    WaveCloseWriteFile(&hmmio, &mmckinfoData,
          &mmckinfoParent, &mmioinfo,
          dwTotalBytesWritten / (wfx.wBitsPerSample / 8));
    return TRUE;
}
```

This function stops the capture buffer, calls the StreamToFile function one more time in order to save all the data up to the current read position, and closes the file. The WaveCloseWriteFile function in Wave.c also updates the header of the data chunk by writing the total number of samples.

Step 9: Shutting Down DirectSoundCapture

[This is preliminary documentation and subject to change.]

Before closing the application you must shut down the capture system. This is a simple matter of releasing all the objects. You must release the **IDirectSoundNotify** interface before releasing the capture buffer.

```
void CleanupDSoundCapture(void)
{
   if (lpdsNotify)
      IDirectSoundNotify_Release(lpdsNotify);
   if (lpdscb)
      IDirectSoundCaptureBuffer_Release(lpdscb);
   if (lpdsc)
      IDirectSoundCapture_Release(lpdsc);
}
```

DirectSound Reference

[This is preliminary documentation and subject to change.]

This section contains reference information for the API elements that DirectSound provides. Reference material is divided into the following categories.

- Interfaces
- Functions
- Callback Function
- Structures
- Return Values

Interfaces

[This is preliminary documentation and subject to change.]

This section contains references for methods of the following DirectSound interfaces:

- **IDirectSound**
- **IDirectSound3DBuffer**
- **IDirectSound3DListener**
- **IDirectSoundBuffer**
- **IDirectSoundCapture**
- **IDirectSoundCaptureBuffer**
- **IDirectSoundNotify**

IDirectSound

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectSound** interface to create DirectSound objects and set up the environment. This section is a reference to the methods of this interface.

The interface is obtained by using the **DirectSoundCreate** function.

The methods of the **IDirectSound** interface can be organized into the following groups:

Initialization	**Initialize** **SetCooperativeLevel**
Buffer creation	**CreateSoundBuffer** **DuplicateSoundBuffer**
Device capabilities	**GetCaps**
Memory management	**Compact**
Speaker configuration	**GetSpeakerConfig** **SetSpeakerConfig**

The **IDirectSound** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef

QueryInterface

Release

The **LPDIRECTSOUND** type is defined as a pointer to the **IDirectSound** interface:

```
typedef struct IDirectSound    *LPDIRECTSOUND;
```

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound::Compact

[This is preliminary documentation and subject to change.]

The **IDirectSound::Compact** method moves the unused portions of on-board sound memory, if any, to a contiguous block so that the largest portion of free memory will be available.

```
HRESULT Compact();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED
DSERR_UNINITIALIZED

Remarks

The application's DirectSound object must have at least the DSSCL_PRIORITY cooperative level. See **IDirectSound::SetCooperativeLevel**.

This method will fail if any operations are in progress.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound::CreateSoundBuffer

[This is preliminary documentation and subject to change.]

The **IDirectSound::CreateSoundBuffer** method creates a DirectSoundBuffer object to hold a sequence of audio samples.

```
HRESULT CreateSoundBuffer(
  LPCDSBUFFERDESC lpcDSBufferDesc,
  LPLPDIRECTSOUNDBUFFER lplpDirectSoundBuffer,
  IUnknown FAR * pUnkOuter
);
```

Parameters

lpcDSBufferDesc
Address of a **DSBUFFERDESC** structure that contains the description of the sound buffer to be created.

lplpDirectSoundBuffer
Address of a pointer to the new DirectSoundBuffer object, or NULL if the buffer cannot be created.

pUnkOuter
Controlling unknown of the aggregate. Its value must be NULL.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_ALLOCATED
DSERR_BADFORMAT
DSERR_INVALIDPARAM
DSERR_NOAGGREGATION
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

Remarks

Before it can play any sound buffers, the application must specify a cooperative level for a DirectSound object by using the **IDirectSound::SetCooperativeLevel** method.

The *lpDSBufferDesc* parameter points to a structure that describes the type of buffer desired, including format, size, and capabilities. The application must specify the needed capabilities, or they will not be available. For example, if the application creates a DirectSoundBuffer object without specifying the DSBCAPS_CTRLFREQUENCY flag, any call to **IDirectSoundBuffer::SetFrequency** will fail.

The DSBCAPS_STATIC flag can also be specified, in which case DirectSound stores the buffer in on-board memory, if available, to take advantage of hardware mixing. To force the buffer to use either hardware or software mixing, use the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

DSBUFFERDESC, **IDirectSound::DuplicateSoundBuffer**, **IDirectSound::SetCooperativeLevel**, **IDirectSoundBuffer**, **IDirectSoundBuffer::GetFormat**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::Lock**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::SetFormat**, **IDirectSoundBuffer::SetFrequency**

IDirectSound::DuplicateSoundBuffer

[This is preliminary documentation and subject to change.]

The **IDirectSound::DuplicateSoundBuffer** method creates a new DirectSoundBuffer object that uses the same buffer memory as the original object.

```
HRESULT DuplicateSoundBuffer(
  LPDIRECTSOUNDBUFFER lpDsbOriginal,
  LPLPDIRECTSOUNDBUFFER lplpDsbDuplicate
);
```

Parameters

lpDsbOriginal
Address of the DirectSoundBuffer object to be duplicated.

lplpDsbDuplicate
Address of a pointer to the new DirectSoundBuffer object.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_ALLOCATED
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

Remarks

The new object can be used just like the original.

Initially, the duplicate buffer will have the same parameters as the original buffer. However, the application can change the parameters of each buffer independently, and each can be played or stopped without affecting the other.

If data in the buffer is changed through one object, the change will be reflected in the other object because the buffer memory is shared.

The buffer memory will be released when the last object referencing it is released.

Applications cannot assume that an attempt to duplicate a sound buffer will always succeed. In particular, DirectSound will not create a software duplicate of a hardware buffer.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound::CreateSoundBuffer

IDirectSound::GetCaps

[This is preliminary documentation and subject to change.]

The **IDirectSound::GetCaps** method retrieves the capabilities of the hardware device that is represented by the DirectSound object.

```
HRESULT GetCaps(
  LPDSCAPS lpDSCaps
);
```

Parameters

lpDSCaps
Address of the **DSCAPS** structure to contain the capabilities of this sound device.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

Information retrieved in the **DSCAPS** structure describes the maximum capabilities of the sound device and those currently available, such as the number of hardware mixing channels and the amount of on-board sound memory. You can use this information to fine-tune performance and optimize resource allocation.

Because of resource-sharing requirements, the maximum capabilities in one area might be available only at the cost of another area. For example, the maximum number of hardware-mixed streaming sound buffers might be available only if there are no hardware static sound buffers.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

DirectSoundCreate, **DSCAPS**

IDirectSound::GetSpeakerConfig

[This is preliminary documentation and subject to change.]

The **IDirectSound::GetSpeakerConfig** method retrieves the speaker configuration specified for this DirectSound object.

```
HRESULT GetSpeakerConfig(
  LPDWORD lpdwSpeakerConfig
);
```

Parameters

lpdwSpeakerConfig
Address of the speaker configuration for this DirectSound object. The speaker configuration is specified with one of the following values:

DSSPEAKER_HEADPHONE
The audio is played through headphones.

DSSPEAKER_MONO
The audio is played through a single speaker.

DSSPEAKER_QUAD
The audio is played through quadraphonic speakers.

DSSPEAKER_STEREO
The audio is played through stereo speakers (default value).

DSSPEAKER_SURROUND
The audio is played through surround speakers.

DSSPEAKER_STEREO may be combined with one of the following values:

DSSPEAKER_GEOMETRY_WIDE
The speakers are directed over an arc of 20 degrees.

DSSPEAKER_GEOMETRY_NARROW
The speakers are directed over an arc of 10 degrees.

DSSPEAKER_GEOMETRY_MIN
The speakers are directed over an arc of 5 degrees.

DSSPEAKER_GEOMETRY_MAX
The speakers are directed over an arc of 180 degrees.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

The value returned at *lpdwSpeakerConfig* may be a packed **DWORD** containing both configuration and geometry information. Use the **DSSPEAKER_CONFIG** and **DSSPEAKER_GEOMETRY** macros to unpack the **DWORD**, as in the following example:

```
if (DSSPEAKER_CONFIG(dwSpeakerConfig) == DSSPEAKER_STEREO)
  {
  if (DSSPEAKER_GEOMETRY(dwSpeakerConfig) ==
           DSSPEAKER_GEOMETRY_WIDE)
    {...}
```

```
}
```

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound::SetSpeakerConfig

IDirectSound::Initialize

[This is preliminary documentation and subject to change.]

The **IDirectSound::Initialize** method initializes the DirectSound object that was created by using the **CoCreateInstance** function.

HRESULT Initialize(
  LPGUID *lpGuid*
);

Parameters

lpGuid
: Address of the globally unique identifier (GUID) specifying the sound driver to which this DirectSound object binds. Pass NULL to select the primary sound driver.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_ALREADYINITIALIZED
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_NODRIVER

Remarks

This method is provided for compliance with the Component Object Model (COM) protocol. If the **DirectSoundCreate** function was used to create the DirectSound object, this method returns DSERR_ALREADYINITIALIZED. If **IDirectSound::Initialize** is not called when using **CoCreateInstance** to create the DirectSound object, any method called afterward returns DSERR_UNINITIALIZED.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

DirectSoundCreate

IDirectSound::SetCooperativeLevel

[This is preliminary documentation and subject to change.]

The **IDirectSound::SetCooperativeLevel** method sets the cooperative level of the application for this sound device.

```
HRESULT SetCooperativeLevel(
  HWND hwnd,
  DWORD dwLevel
);
```

Parameters

hwnd
: Window handle to the application.

dwLevel
: Requested priority level. Specify one of the following values:

DSSCL_EXCLUSIVE
: Sets the application to the exclusive level. When it has the input focus, the application will be the only one audible (sounds from applications with the DSBCAPS_GLOBALFOCUS flag set will be muted). With this level, it also has all the privileges of the DSSCL_PRIORITY level. DirectSound will restore the hardware format, as specified by the most recent call to the

IDirectSoundBuffer::SetFormat method, once the application gains the input focus. (Note that DirectSound will always restore the wave format no matter what priority level is set.)

DSSCL_NORMAL
: Sets the application to a fully cooperative status. This level has the smoothest multitasking and resource-sharing behavior, but because it does not allow the primary buffer format to change, output is restricted to the default 8-bit format.

DSSCL_PRIORITY
: Sets the application to the priority level. Applications with this cooperative level can call the **IDirectSoundBuffer::SetFormat** and **IDirectSound::Compact** methods.

DSSCL_WRITEPRIMARY
: This is the highest priority level. The application has write access to the primary sound buffers. No secondary sound buffers can be played. This level cannot be set if the DirectSound driver is being emulated for the device; that is, if the **IDirectSound::GetCaps** method returns the DSCAPS_EMULDRIVER flag in the **DSCAPS** structure.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_ALLOCATED
DSERR_INVALIDPARAM
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

Remarks

The application must set the cooperative level by calling this method before its buffers can be played. The recommended cooperative level is DSSCL_PRIORITY; use other priority levels when necessary. For additional information, see Cooperative Levels.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound::Compact, **IDirectSoundBuffer::GetFormat**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::Lock**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::Restore**, **IDirectSoundBuffer::SetFormat**

IDirectSound::SetSpeakerConfig

[This is preliminary documentation and subject to change.]

The **IDirectSound::SetSpeakerConfig** method specifies the speaker configuration of the DirectSound object.

```
HRESULT SetSpeakerConfig(
  DWORD dwSpeakerConfig
);
```

Parameters

dwSpeakerConfig
: Speaker configuration of the specified DirectSound object. This parameter can be one of the following values:

DSSPEAKER_HEADPHONE
: The speakers are headphones.

DSSPEAKER_MONO
: The speakers are monaural.

DSSPEAKER_QUAD
: The speakers are quadraphonic.

DSSPEAKER_STEREO
: The speakers are stereo (default value).

DSSPEAKER_SURROUND
: The speakers are surround sound.

DSSPEAKER_STEREO may be combined with one of the following values:

DSSPEAKER_GEOMETRY_WIDE
: The speakers are directed over an arc of 20 degrees.

DSSPEAKER_GEOMETRY_NARROW
: The speakers are directed over an arc of 10 degrees.

DSSPEAKER_GEOMETRY_MIN
: The speakers are directed over an arc of 5 degrees.

DSSPEAKER_GEOMETRY_MAX
: The speakers are directed over an arc of 180 degrees.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

If a geometry value is to be used, it must be packed in a **DWORD** along with the DSSPEAKER_STEREO flag. This can be done by using the **DSSPEAKER_COMBINED** macro, as in the following C++ example:

```
lpds->SetSpeakerConfig(DSSPEAKER_COMBINED(
    DSSPEAKER_STEREO, DSSPEAKER_GEOMETRY_WIDE));
```

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound::GetSpeakerConfig

IDirectSound3DBuffer

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectSound3DBuffer** interface to retrieve and set parameters that describe the position, orientation, and environment of a sound buffer in 3-D space. This section is a reference to the methods of this interface. For a conceptual overview, see DirectSound 3-D Buffers.

The **IDirectSound3DBuffer** is obtaining by using the **IDirectSoundBuffer::QueryInterface** method. For more information, see Obtaining the IDirectSound3DBuffer Interface.

The methods of the **IDirectSound3DBuffer** interface can be organized into the following groups:

Batch parameter manipulation	**GetAllParameters**
	SetAllParameters

Distance	**GetMaxDistance** **GetMinDistance** **SetMaxDistance** **SetMinDistance**
Operation mode	**GetMode** **SetMode**
Position	**GetPosition** **SetPosition**
Sound projection cones	**GetConeAngles** **GetConeOrientation** **GetConeOutsideVolume** **SetConeAngles** **SetConeOrientation** **SetConeOutsideVolume**
Velocity	**GetVelocity** **SetVelocity**

The **IDirectSound3DBuffer** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECTSOUND3DBUFFER** type is defined as a pointer to the **IDirectSound3DBuffer** interface:

```
typedef struct IDirectSound3DBuffer    *LPDIRECTSOUND3DBUFFER;
```

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.

Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DBuffer::GetAllParameters

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::GetAllParameters** method retrieves information that describes the 3-D characteristics of a sound buffer at a given point in time.

```
HRESULT GetAllParameters(
  LPDS3DBUFFER lpDs3dBuffer
);
```

Parameters

lpDs3dBuffer
Address of a **DS3DBUFFER** structure that will contain the information describing the 3-D characteristics of the sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DBuffer::GetConeAngles

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::GetConeAngles** method retrieves the inside and outside angles of the sound projection cone for this sound buffer.

```
HRESULT GetConeAngles(
  LPDWORD lpdwInsideConeAngle,
  LPDWORD lpdwOutsideConeAngle
```

```
);
```

Parameters

lpdwInsideConeAngle and *lpdwOutsideConeAngle*
: Addresses of variables that will contain the inside and outside angles of the sound projection cone, in degrees.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The minimum, maximum, and default cone angles are defined in Dsound.h as DS3D_MINCONEANGLE, DS3D_MAXCONEANGLE, and DS3D_DEFAULTCONEANGLE.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DBuffer::GetConeOrientation

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::GetConeOrientation** method retrieves the orientation of the sound projection cone for this sound buffer.

```
HRESULT GetConeOrientation(
  LPD3DVECTOR lpvOrientation
);
```

Parameters

lpvOrientation
: Address of a **D3DVECTOR** structure that will contain the current orientation of the sound projection cone. The vector information represents the center of the sound cone.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The values returned are not necessarily the same as those set by using the **IDirectSound3DBuffer::SetConeOrientation** method. DirectSound adjusts orientation vectors so that they are have a magnitude of less than or equal to 1.0.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DBuffer::SetConeOrientation, **IDirectSound3DBuffer::SetConeAngles**, **IDirectSound3DBuffer::SetConeOutsideVolume**

IDirectSound3DBuffer::GetConeOutsideVolume

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::GetConeOutsideVolume** method retrieves the current cone outside volume for this sound buffer.

```
HRESULT GetConeOutsideVolume(
  LPLONG lplConeOutsideVolume
);
```

Parameters

lplConeOutsideVolume
Address of a variable that will contain the current cone outside volume for this buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Volume levels are represented by attenuation. Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). The default value is DS3D_DEFAULTCONEOUTSIDEVOLUME (no attenuation). These values are defined in Dsound.h. Currently DirectSound does not support amplification.

For additional information about the concept of outside volume, see Sound Cones.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer::SetVolume

IDirectSound3DBuffer::GetMaxDistance

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::GetMaxDistance** method retrieves the current maximum distance for this sound buffer.

```
HRESULT GetMaxDistance(
  LPD3DVALUE lpflMaxDistance
);
```

Parameters

lpflMaxDistance
Address of a variable that will contain the current maximum distance setting.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The default maximum distance, defined as DS3D_DEFAULTMAXDISTANCE, is effectively infinite.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DBuffer::GetMinDistance, **IDirectSound3DBuffer::SetMaxDistance**

IDirectSound3DBuffer::GetMinDistance

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::GetMinDistance** method retrieves the current minimum distance for this sound buffer.

```
HRESULT GetMinDistance(
  LPD3DVALUE lpflMinDistance
);
```

Parameters

lpflMinDistance
Address of a variable that will contain the current minimum distance setting.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

By default, the minimum distance value is DS3D_DEFAULTMINDISTANCE, currently defined as 1.0 (corresponding to 1.0 meter at the default distance factor of 1.0 meters per unit).

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DBuffer::SetMinDistance, **IDirectSound3DBuffer::GetMaxDistance**

IDirectSound3DBuffer::GetMode

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::GetMode** method retrieves the current operation mode for 3-D sound processing.

```
HRESULT GetMode(
  LPDWORD lpdwMode
);
```

Parameters

lpdwMode
: Address of a variable that will contain the current mode setting. This value will be one of the following:

  DS3DMODE_DISABLE
  : Processing of 3-D sound is disabled. The sound seems to originate from the center of the listener's head.

  DS3DMODE_HEADRELATIVE
  : Sound parameters (position, velocity, and orientation) are relative to the listener's parameters. In this mode, the absolute parameters of the sound are updated automatically as the listener's parameters change, so that the relative parameters remain constant.

  DS3DMODE_NORMAL
  : Normal processing. This is the default mode.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DBuffer::GetPosition

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::GetPosition** method retrieves the sound buffer's current position, in distance units. By default, distance units are meters, but the units can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT GetPosition(
  LPD3DVECTOR lpvPosition
);
```

Parameters

lpvPosition
: Address of a **D3DVECTOR** structure that will contain the current position of the sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DBuffer::GetVelocity

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::GetVelocity** method retrieves the current velocity for this sound buffer. Velocity is measured in units per second. The default unit is one meter, but this can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT GetVelocity(
  LPD3DVECTOR lpvVelocity
);
```

Parameters

lpvVelocity
: Address of a **D3DVECTOR** structure that will contain the sound buffer's current velocity.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Velocity is used for Doppler effects only. It does not actually move the buffer. For additional information, see Doppler Factor.

The default unit of measurement is meters per second, but this can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DBuffer::SetPosition, **IDirectSound3DBuffer::SetVelocity**

IDirectSound3DBuffer::SetAllParameters

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::SetAllParameters** method sets all 3-D sound buffer parameters from a given **DS3DBUFFER** structure that describes all aspects of the sound buffer at a moment in time.

```
HRESULT SetAllParameters(
  LPCDS3DBUFFER lpcDs3dBuffer,
  DWORD dwApply
);
```

Parameters

lpcDs3dBuffer
: Address of a **DS3DBUFFER** structure containing the information that describes the 3-D characteristics of the sound buffer.

dwApply
: Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DBuffer::SetConeAngles

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::SetConeAngles** method sets the inside and outside angles of the sound projection cone for this sound buffer.

```
HRESULT SetConeAngles(
  DWORD dwInsideConeAngle,
  DWORD dwOutsideConeAngle,
  DWORD dwApply
);
```

Parameters

dwInsideConeAngle and *dwOutsideConeAngle*
Inside and outside angles of the sound projection cone, in degrees.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The minimum, maximum, and default cone angles are defined in Dsound.h as DS3D_MINCONEANGLE, DS3D_MAXCONEANGLE, and DS3D_DEFAULTCONEANGLE. Each angle must be in the range of 0 degrees (no cone) to 360 degrees (the full sphere). The default value is 360.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

I**IDirectSound3DBuffer::GetConeOutsideVolume**, **IDirectSound3DBuffer::SetConeOutsideVolume**

IDirectSound3DBuffer::SetConeOrientation

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::SetConeOrientation** method sets the orientation of the sound projection cone for this sound buffer. This method has no effect unless the cone angle and cone volume factor have also been set.

```
HRESULT SetConeOrientation(
  D3DVALUE x,
  D3DVALUE y,
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*
Values whose types are **D3DVALUE** and that represent the coordinates of the new sound cone orientation vector.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DBuffer::SetConeAngles, **IDirectSound3DBuffer::SetConeOutsideVolume**

IDirectSound3DBuffer::SetConeOutsideVolume

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::SetConeOutsideVolume** method sets the current cone outside volume for this sound buffer.

```
HRESULT SetConeOutsideVolume(
  LONG lConeOutsideVolume,
  DWORD dwApply
);
```

Parameters

lConeOutsideVolume
Cone outside volume for this sound buffer, in hundredths of decibels. Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are defined in Dsound.h.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change

	several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Volume levels are represented by attenuation. Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). The default value is DS3D_DEFAULTCONEOUTSIDEVOLUME (no attenuation). These values are defined in Dsound.h. Currently DirectSound does not support amplification.

For information about the concept of cone outside volume, see Sound Cones.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer::SetVolume

IDirectSound3DBuffer::SetMaxDistance

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::SetMaxDistance** method sets the current maximum distance value.

```
HRESULT SetMaxDistance(
  D3DVALUE flMaxDistance,
  DWORD dwApply
);
```

Parameters

flMaxDistance
New maximum distance value.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The default maximum distance, defined as DS3D_DEFAULTMAXDISTANCE, is effectively infinite.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DBuffer::GetMaxDistance, **IDirectSound3DBuffer::SetMinDistance**

IDirectSound3DBuffer::SetMinDistance

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::SetMinDistance** method sets the current minimum distance value.

```
HRESULT SetMinDistance(
  D3DVALUE flMinDistance,
  DWORD dwApply
);
```

Parameters

flMinDistance
New minimum distance value.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE
Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

By default, the minimum distance value is DS3D_DEFAULTMINDISTANCE, currently defined as 1.0 (corresponding to 1.0 meter at the default distance factor of 1.0 meters per unit).

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DBuffer::SetMaxDistance

IDirectSound3DBuffer::SetMode

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::SetMode** method sets the operation mode for 3-D sound processing.

```
HRESULT SetMode(
  DWORD dwMode,
  DWORD dwApply
);
```

Parameters

dwMode

Flag specifying the 3-D sound processing mode to be set:

DS3DMODE_DISABLE
: Processing of 3-D sound is disabled. The sound seems to originate from the center of the listener's head.

DS3DMODE_HEADRELATIVE
: Sound parameters (position, velocity, and orientation) are relative to the listener's parameters. In this mode, the absolute parameters of the sound are updated automatically as the listener's parameters change, so that the relative parameters remain constant.

DS3DMODE_NORMAL
: Normal processing. This is the default mode.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED
: Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE
: Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DBuffer::SetPosition

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::SetPosition** method sets the sound buffer's current position, in distance units. By default, distance units are meters, but the units can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT SetPosition(
  D3DVALUE x,
  D3DVALUE y,
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*
: Values whose types are **D3DVALUE** and that represent the coordinates of the new position vector. Note that DirectSound may adjust these values to prevent floating-point overflow.

dwApply
: Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED
: Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE
: Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DBuffer::SetVelocity

[This is preliminary documentation and subject to change.]

The **IDirectSound3DBuffer::SetVelocity** method sets the sound buffer's current velocity.

```
HRESULT SetVelocity(
  D3DVALUE x,
  D3DVALUE y,
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*
: Values whose types are **D3DVALUE** and that represent the coordinates of the new velocity vector. Note that DirectSound may adjust these values to prevent floating-point overflow.

dwApply
: Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED
: Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE
: Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Velocity is used for Doppler effects only. It does not actually move the buffer. For additional information, see Doppler Factor.

The default unit of measurement is meters per second, but this can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DBuffer::SetPosition, **IDirectSound3DBuffer::GetVelocity**

IDirectSound3DListener

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectSound3DListener** interface to retrieve and set parameters that describe a listener's position, orientation, and listening environment in 3-D space. This section is a reference to the methods of this interface. For a conceptual overview, see DirectSound 3-D Listeners.

The interface is obtained by using the **IDirectSoundBuffer::QueryInterface** method. For more information, see Obtaining the IDirectSound3DListener Interface.

The methods of the **IDirectSound3DListener** interface can be organized into the following groups:

Batch parameters	**GetAllParameters** **SetAllParameters**
Deferred settings	**CommitDeferredSettings**
Distance factor	**GetDistanceFactor** **SetDistanceFactor**
Doppler factor	**GetDopplerFactor** **SetDopplerFactor**

Orientation	**GetOrientation** **SetOrientation**
Position	**GetPosition** **SetPosition**
Rolloff factor	**GetRolloffFactor** **SetRolloffFactor**
Velocity	**GetVelocity** **SetVelocity**

The **IDirectSound3DListener** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef

QueryInterface

Release

The **LPDIRECTSOUND3DLISTENER** type is defined as a pointer to the **IDirectSound3DListener** interface:

```
typedef struct IDirectSound3DListener    *LPDIRECTSOUND3DLISTENER;
```

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DListener::CommitDeferredSettings

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::CommitDeferredSettings** method commits any deferred settings made since the last call to this method.

```
HRESULT CommitDeferredSettings();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

For additional information about using deferred settings to maximize efficiency, see Deferred Settings.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSound3DListener::GetAllParameters

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::GetAllParameters** method retrieves information that describes the current state of the 3-D world and listener.

```
HRESULT GetAllParameters(
  LPDS3DLISTENER lpListener
);
```

Parameters

lpListener
: Address of a **DS3DLISTENER** structure that will contain the current state of the 3-D world and listener.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::SetAllParameters

IDirectSound3DListener::GetDistanceFactor

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::GetDistanceFactor** method retrieves the current distance factor.

```
HRESULT GetDistanceFactor(
  LPD3DVALUE lpflDistanceFactor
);
```

Parameters

lpflDistanceFactor
: Address of a variable whose type is **D3DVALUE** and that will contain the current distance factor value.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

For additional information about distance factors, see Distance Factor.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.

Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::SetDistanceFactor

IDirectSound3DListener::GetDopplerFactor

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::GetDopplerFactor** method retrieves the current Doppler effect factor.

```
HRESULT GetDopplerFactor(
  LPD3DVALUE lpflDopplerFactor
);
```

Parameters

lpflDopplerFactor
: Address of a variable whose type is **D3DVALUE** and that will contain the current Doppler factor value.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The Doppler factor has a range of DS3D_MINDOPPLERFACTOR (no Doppler effects) to DS3D_MAXDOPPLERFACTOR (as currently defined, 10 times the Doppler effects found in the real world). The default value is DS3D_DEFAULTDOPPLERFACTOR (1.0). For additional information, see Doppler Factor.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::SetDopplerFactor

IDirectSound3DListener::GetOrientation

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::GetOrientation** method retrieves the listener's current orientation in vectors: a front vector and a top vector.

```
HRESULT GetOrientation(
  LPD3DVECTOR lpvOrientFront,
  LPD3DVECTOR lpvOrientTop
);
```

Parameters

lpvOrientFront
: Address of a **D3DVECTOR** structure that will contain the listener's front orientation vector.

lpvOrientTop
: Address of a **D3DVECTOR** structure that will contain the listener's top orientation vector.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The front vector points in the direction of the listener's nose, and the top vector points out the top of the listener's head. By default, the front vector is (0,0,1.0) and the top vector is (0,1.0,0).

The values returned are not necessarily the same as those set by using the **IDirectSound3DListener::SetOrientation** method. DirectSound adjusts orientation vectors so that they are at right angles and have a magnitude of less than or equal to 1.0.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::SetOrientation

IDirectSound3DListener::GetPosition

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::GetPosition** method retrieves the listener's current position in distance units. By default, these units are meters, but this can be changed by calling the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT GetPosition(
  LPD3DVECTOR lpvPosition
);
```

Parameters

lpvPosition
Address of a **D3DVECTOR** structure that will contain the listener's position vector.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::SetPosition

IDirectSound3DListener::GetRolloffFactor

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::GetRolloffFactor** method retrieves the current rolloff factor.

```
HRESULT GetRolloffFactor(
  LPD3DVALUE lpflRolloffFactor
);
```

Parameters

lpflRolloffFactor
Address of a variable whose type is **D3DVALUE** and that will contain the current rolloff factor value.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The rolloff factor has a range of DS3D_MINROLLOFFFACTOR (no rolloff) to DS3D_MAXROLLOFFFACTOR (as currently defined, 10 times the rolloff found in the real world). The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0). For additional information, see Rolloff Factor.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::SetRolloffFactor

IDirectSound3DListener::GetVelocity

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::GetVelocity** method retrieves the listener's current velocity.

```
HRESULT GetVelocity(
  LPD3DVECTOR lpvVelocity
);
```

Parameters

lpvVelocity
: Address of a **D3DVECTOR** structure that will contain the listener's current velocity.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Velocity is used only for Doppler effects. It does not actually move the listener. To change the listener's position, use the **IDirectSound3DListener::SetPosition** method. The default velocity is (0,0,0).

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::SetVelocity

IDirectSound3DListener::SetAllParameters

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::SetAllParameters** method sets all 3-D listener parameters from a given **DS3DLISTENER** structure that describes all aspects of the 3-D listener at a moment in time.

```
HRESULT SetAllParameters(
  LPCDS3DLISTENER lpcListener,
  DWORD dwApply
);
```

Parameters

lpcListener
: Address of a **DS3DLISTENER** structure that contains information describing all current 3-D listener parameters.

dwApply
: Value indicating when the setting should be applied. This value must be one of the following:

  DS3D_DEFERRED
  : Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

  DS3D_IMMEDIATE
  : Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::GetAllParameters

IDirectSound3DListener::SetDistanceFactor

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::SetDistanceFactor** method sets the current distance factor.

```
HRESULT SetDistanceFactor(
  D3DVALUE flDistanceFactor,
  DWORD dwApply
);
```

Parameters

flDistanceFactor
: New distance factor.

dwApply
: Value indicating when the setting should be applied. This value must be one of the following:

  DS3D_DEFERRED
  : Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

  DS3D_IMMEDIATE
  : Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

For additional information about distance factors, see Distance Factor.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::GetDistanceFactor

IDirectSound3DListener::SetDopplerFactor

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::SetDopplerFactor** method sets the current Doppler effect factor.

```
HRESULT SetDopplerFactor(
  D3DVALUE flDopplerFactor,
  DWORD dwApply
);
```

Parameters

flDopplerFactor
New Doppler factor value.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE
Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The Doppler factor has a range of DS3D_MINDOPPLERFACTOR (no Doppler effects) to DS3D_MAXDOPPLERFACTOR (as currently defined, 10 times the

Doppler effects found in the real world). The default value is DS3D_DEFAULTDOPPLERFACTOR (1.0). For additional information, see Doppler Factor.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::GetDopplerFactor

IDirectSound3DListener::SetOrientation

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::SetOrientation** method sets the listener's current orientation in terms of two vectors: a front vector and a top vector.

```
HRESULT SetOrientation(
  D3DVALUE xFront,
  D3DVALUE yFront,
  D3DVALUE zFront,
  D3DVALUE xTop,
  D3DVALUE yTop,
  D3DVALUE zTop,
  DWORD dwApply
);
```

Parameters

xFront, yFront, and *zFront*
: Values whose types are **D3DVALUE** and that represent the coordinates of the front orientation vector.

xTop, yTop, and *zTop*
: Values whose types are **D3DVALUE** and that represent the coordinates of the top orientation vector.

dwApply
: Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE
Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The front vector points in the direction of the listener's nose, and the top vector points out the top of the listener's head. By default, the front vector is (0,0,1.0) and the top vector is (0,1.0,0).

The top vector must be at right angles to the front vector. If necessary, DirectSound adjusts the front vector after setting the top vector.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::GetOrientation

IDirectSound3DListener::SetPosition

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::SetPosition** method sets the listener's current position, in distance units. By default, these units are meters, but this can be changed by calling the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT SetPosition(
  D3DVALUE x,
  D3DVALUE y,
```

```
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*
: Values whose types are **D3DVALUE** and that represent the coordinates of the listener's new position vector. Note that DirectSound may adjust these values to prevent floating-point overflow.

dwApply
: Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED
: Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE
: Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::GetPosition

IDirectSound3DListener::SetRolloffFactor

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::SetRolloffFactor** method sets the rolloff factor.

```
HRESULT SetRolloffFactor(
  D3DVALUE flRolloffFactor,
  DWORD dwApply
);
```

Parameters

flRolloffFactor
New rolloff factor.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE
Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The rolloff factor has a range of DS3D_MINROLLOFFFACTOR (no rolloff) to DS3D_MAXROLLOFFFACTOR (as currently defined, 10 times the rolloff found in the real world). The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0). For additional information, see Rolloff Factor.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::GetRolloffFactor

IDirectSound3DListener::SetVelocity

[This is preliminary documentation and subject to change.]

The **IDirectSound3DListener::SetVelocity** method sets the listener's velocity.

```
HRESULT SetVelocity(
  D3DVALUE x,
  D3DVALUE y,
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*
: Values whose types are **D3DVALUE** and that represent the coordinates of the listener's new velocity vector. Note that DirectSound may adjust these values to prevent floating-point overflow.

dwApply
: Value indicating when the setting should be applied. This value must be one of the following:

  DS3D_DEFERRED
  : Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

  DS3D_IMMEDIATE
  : Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Velocity is used only for Doppler effects. It does not actually move the listener. To change the listener's position, use the **IDirectSound3DListener::SetPosition** method. The default velocity is (0,0,0).

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound3DListener::GetVelocity

IDirectSoundBuffer

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectSoundBuffer** interface to create DirectSoundBuffer objects and set up the environment.

The interface is obtained by using the **IDirectSound::CreateSoundBuffer** method.

The **IDirectSoundBuffer** methods can be organized into the following groups:

Information	**GetCaps**
	GetFormat
	GetStatus
	SetFormat
Memory management	**Initialize**
	Restore
Play management	**GetCurrentPosition**
	Lock
	Play
	SetCurrentPosition
	Stop
	Unlock
Sound management	**GetFrequency**
	GetPan
	GetVolume
	SetFrequency
	SetPan

SetVolume

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

AddRef

QueryInterface

Release

The **LPDIRECTSOUNDBUFFER** type is defined as a pointer to the **IDirectSoundBuffer** interface:

```
typedef struct IDirectSoundBuffer    *LPDIRECTSOUNDBUFFER;
```

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundBuffer::GetCaps

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::GetCaps** method retrieves the capabilities of the DirectSoundBuffer object.

```
HRESULT GetCaps(
  LPDSBCAPS lpDSBufferCaps
);
```

Parameters

lpDSBufferCaps
Address of a **DSBCAPS** structure to contain the capabilities of this sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The **DSBCAPS** structure contains similar information to the **DSBUFFERDESC** structure passed to the **IDirectSound::CreateSoundBuffer** method, with some additional information. This additional information can include the buffer's location, either in hardware or software, and some cost measures. Examples of cost measures include the time it takes to download to a hardware buffer and the processing overhead required to mix and play the buffer when it is in the system memory.

The flags specified in the **dwFlags** member of the **DSBCAPS** structure are the same flags used by the **DSBUFFERDESC** structure. The only difference is that in the **DSBCAPS** structure, either DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE will be specified according to the location of the buffer memory. In the **DSBUFFERDESC** structure, these flags are optional and, depending on which flag is specified, force the buffer to be located in either hardware or software.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

DSBCAPS, **DSBUFFERDESC**, **IDirectSoundBuffer**, **IDirectSound::CreateSoundBuffer**

IDirectSoundBuffer::GetCurrentPosition

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::GetCurrentPosition** method retrieves the current position of the play and write cursors in the sound buffer.

```
HRESULT GetCurrentPosition(
  LPDWORD lpdwCurrentPlayCursor,
  LPDWORD lpdwCurrentWriteCursor
);
```

Parameters

lpdwCurrentPlayCursor

Address of a variable to contain the current play position in the DirectSoundBuffer object. This position is an offset within the sound buffer and is specified in bytes. This parameter can be NULL if the current play position is not wanted.

lpdwCurrentWriteCursor

Address of a variable to contain the current write position in the DirectSoundBuffer object. This position is an offset within the sound buffer and is specified in bytes. This parameter can be NULL if the current write position is not wanted.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM

DSERR_PRIOLEVELNEEDED

Remarks

The write cursor indicates the position at which it is safe to write new data to the buffer. The write cursor always leads the play cursor, typically by about 15 milliseconds' worth of audio data. For more information, see Current Play and Write Positions.

It is always safe to change data that is behind the position indicated by the *lpdwCurrentPlayCursor* parameter.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetCurrentPosition**

IDirectSoundBuffer::GetFormat

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::GetFormat** method retrieves a description of the format of the sound data in the buffer, or the buffer size needed to retrieve the format description.

```
HRESULT GetFormat(
  LPWAVEFORMATEX lpwfxFormat,
  DWORD dwSizeAllocated,
  LPDWORD lpdwSizeWritten
);
```

Parameters

lpwfxFormat
: Address of the **WAVEFORMATEX** structure to contain a description of the sound data in the buffer. To retrieve the buffer size needed to contain the format description, specify NULL. In this case the **DWORD** pointed to by the *lpdwSizeWritten* parameter will receive the size of the structure needed to receive complete format information.

dwSizeAllocated
: Size, in bytes, of the **WAVEFORMATEX** structure. DirectSound writes, at most, *dwSizeAllocated* bytes to that pointer; if the **WAVEFORMATEX** structure requires more memory, it is truncated.

lpdwSizeWritten
: Address of a variable to contain the number of bytes written to the **WAVEFORMATEX** structure. This parameter can be NULL.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The **WAVEFORMATEX** structure can have a variable length that depends on the details of the format. Before retrieving the format description, the application should query the DirectSoundBuffer object for the size of the format by calling this method and specifying NULL for the *lpwfxFormat* parameter. The size of the structure will be returned in the *lpdwSizeWritten* parameter. The application can then allocate sufficient memory and call **IDirectSoundBuffer::GetFormat** again to retrieve the format description.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetFormat**

IDirectSoundBuffer::GetFrequency

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::GetFrequency** method retrieves the frequency, in samples per second, at which the buffer is playing.

```
HRESULT GetFrequency(
  LPDWORD lpdwFrequency
);
```

Parameters

lpdwFrequency
Address of the variable that represents the frequency at which the audio buffer is being played.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The frequency value will be in the range of DSBFREQUENCY_MIN to DSBFREQUENCY_MAX. These values are currently defined in Dsound.h as 100 and 100,000 respectively.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.

Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetFrequency**

IDirectSoundBuffer::GetPan

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::GetPan** method retrieves a variable that represents the relative volume between the left and right audio channels.

```
HRESULT GetPan(
  LPLONG lplPan
);
```

Parameters

lplPan
: Address of a variable to contain the relative mix between the left and right speakers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The returned value is measured in hundredths of a decibel (dB), in the range of DSBPAN_LEFT to DSBPAN_RIGHT. These values are currently defined in Dsound.h as -10,000 and 10,000 respectively. The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER, defined as zero. This value of 0 in the *lplPan* parameter means that both channels are at full volume (they are attenuated by 0 decibels). At any setting other than DSBPAN_CENTER, one of the channels is at full volume and the other is attenuated.

A pan of -2173 means that the left channel is at full volume and the right channel is attenuated by 21.73 dB. Similarly, a pan of 870 means that the left channel is

attenuated by 8.7 dB and the right channel is at full volume. A pan of DSBPAN_LEFT means that the right channel is silent and the sound is all the way to the left, while a pan of DSBPAN_RIGHT means that the left channel is silent and the sound is all the way to the right.

The pan control acts cumulatively with the volume control.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetPan**, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::GetStatus

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::GetStatus** method retrieves the current status of the sound buffer.

```
HRESULT GetStatus(
  LPDWORD lpdwStatus
);
```

Parameters

lpdwStatus
: Address of a variable to contain the status of the sound buffer. The status can be a combination of the following flags:

DSBSTATUS_BUFFERLOST
: The buffer is lost and must be restored before it can be played or locked.

DSBSTATUS_LOOPING
: The buffer is being looped. If this value is not set, the buffer will stop when it reaches the end of the sound data. Note that if this value is set, the buffer must also be playing.

DSBSTATUS_PLAYING
: The buffer is playing. If this value is not set, the buffer is stopped.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer

IDirectSoundBuffer::GetVolume

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::GetVolume** method retrieves the current volume for this sound buffer.

```
HRESULT GetVolume(
  LPLONG lplVolume
);
```

Parameters

lplVolume
: Address of the variable to contain the volume associated with the specified DirectSound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The volume is specified in hundredths of decibels (dB). Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are currently defined in Dsound.h as 0 and -10,000 respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the stream. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for all practical purposes, is silence. Currently DirectSound does not support amplification.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::Initialize

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::Initialize** method initializes a DirectSoundBuffer object if it has not yet been initialized.

```
HRESULT Initialize(
  LPDIRECTSOUND lpDirectSound,
  LPCDSBUFFERDESC lpcDSBufferDesc
);
```

Parameters

lpDirectSound
: Address of the DirectSound object associated with this DirectSoundBuffer object.

lpcDSBufferDesc
: Address of a **DSBUFFERDESC** structure that contains the values used to initialize this sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values.

DSERR_INVALIDPARAM
DSERR_ALREADYINITIALIZED

Remarks

Because the **IDirectSound::CreateSoundBuffer** method calls **IDirectSoundBuffer::Initialize** internally, it is not needed for the current release of DirectSound. This method is provided for future extensibility.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

DSBUFFERDESC, **IDirectSound::CreateSoundBuffer**, **IDirectSoundBuffer**

IDirectSoundBuffer::Lock

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::Lock** method obtains a valid write pointer to the sound buffer's audio data.

```
HRESULT Lock(
  DWORD dwWriteCursor,
  DWORD dwWriteBytes,
  LPVOID lplpvAudioPtr1,
  LPDWORD lpdwAudioBytes1,
  LPVOID lplpvAudioPtr2,
  LPDWORD lpdwAudioBytes2,
  DWORD dwFlags
);
```

Parameters

dwWriteCursor
: Offset, in bytes, from the start of the buffer to where the lock begins. This parameter is ignored if DSBLOCK_FROMWRITECURSOR is specified in the *dwFlags* parameter.

dwWriteBytes
Size, in bytes, of the portion of the buffer to lock. Note that the sound buffer is conceptually circular.

lplpvAudioPtr1
Address of a pointer to contain the first block of the sound buffer to be locked.

lpdwAudioBytes1
Address of a variable to contain the number of bytes pointed to by the *lplpvAudioPtr1* parameter. If this value is less than the *dwWriteBytes* parameter, *lplpvAudioPtr2* will point to a second block of sound data.

lplpvAudioPtr2
Address of a pointer to contain the second block of the sound buffer to be locked. If the value of this parameter is NULL, the *lplpvAudioPtr1* parameter points to the entire locked portion of the sound buffer.

lpdwAudioBytes2
Address of a variable to contain the number of bytes pointed to by the *lplpvAudioPtr2* parameter. If *lplpvAudioPtr2* is NULL, this value will be 0.

dwFlags
Flags modifying the lock event. The following flags are defined:

DSBLOCK_FROMWRITECURSOR
Locks from the current write position, making a call to **IDirectSoundBuffer::GetCurrentPosition** unnecessary. If this flag is specified, the *dwWriteCursor* parameter is ignored.

DSBLOCK_ENTIREBUFFER
Locks the entire buffer. The *dwWriteBytes* parameter is ignored.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

This method accepts an offset and a byte count, and returns two write pointers and their associated sizes. Two pointers are required because sound buffers are circular. If the locked bytes do not wrap around the end of the buffer, the second pointer, *lplpvAudioBytes2*, will be NULL. However, if the bytes do wrap around, then the second pointer will point to the beginning of the buffer.

If the application passes NULL for the *lplpvAudioPtr2* and *lpdwAudioBytes2* parameters, DirectSound will not lock the wraparound portion of the buffer.

The application should write data to the pointers returned by the **IDirectSoundBuffer::Lock** method, and then call the **IDirectSoundBuffer::Unlock** method to release the buffer back to DirectSound. The sound buffer should not be locked for long periods of time; if it is, the play cursor will reach the locked bytes and configuration-dependent audio problems, possibly random noise, will result.

Warning

This method returns a write pointer only. The application should not try to read sound data from this pointer; the data might not be valid even though the DirectSoundBuffer object contains valid sound data. For example, if the buffer is located in on-board memory, the pointer might be an address to a temporary buffer in main system memory. When **IDirectSoundBuffer::Unlock** is called, this temporary buffer will be transferred to the on-board memory.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Unlock**

IDirectSoundBuffer::Play

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::Play** method causes the sound buffer to play from the current position.

```
HRESULT Play(
  DWORD dwReserved1,
  DWORD dwReserved2,
  DWORD dwFlags
);
```

Parameters

dwReserved1
: This parameter is reserved. Its value must be 0.

dwReserved2
: This parameter is reserved. Its value must be 0.

dwFlags

Flags specifying how to play the buffer. The following flag is defined:

DSBPLAY_LOOPING

Once the end of the audio buffer is reached, play restarts at the beginning of the buffer. Play continues until explicitly stopped. This flag must be set when playing primary sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_BUFFERLOST

DSERR_INVALIDCALL

DSERR_INVALIDPARAM

DSERR_PRIOLEVELNEEDED

Remarks

This method will cause a secondary sound buffer to be mixed into the primary buffer and sent to the sound device. If this is the first buffer to play, it will implicitly create a primary buffer and start playing that buffer; the application need not explicitly direct the primary buffer to play.

If the buffer specified in the method is already playing, the call to the method will succeed and the buffer will continue to play. However, the flags defined in the most recent call supersede flags defined in previous calls.

Primary buffers must be played with the DSBPLAY_LOOPING flag set.

This method will cause primary sound buffers to start playing to the sound device. If the application is set to the DSSCL_WRITEPRIMARY cooperative level, this will cause the audio data in the primary buffer to be sent to the sound device. However, if the application is set to any other cooperative level, this method will ensure that the primary buffer is playing even when no secondary buffers are playing; in that case, silence will be played. This can reduce processing overhead when sounds are started and stopped in sequence, because the primary buffer will be playing continuously rather than stopping and starting between secondary buffers.

Note

Before this method can be called on any sound buffer, the application should call the **IDirectSound::SetCooperativeLevel** method and specify a cooperative level, typically DSSCL_NORMAL. If **IDirectSound::SetCooperativeLevel** has not been called, the **IDirectSoundBuffer::Play** method returns with DS_OK, but no sound will be produced until **IDirectSound::SetCooperativeLevel** is called.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSound::SetCooperativeLevel**

IDirectSoundBuffer::Restore

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::Restore** method restores the memory allocation for a lost sound buffer for the specified DirectSoundBuffer object.

HRESULT Restore();

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

If the application does not have the input focus, **IDirectSoundBuffer::Restore** might not succeed. For example, if the application with the input focus has the DSSCL_WRITEPRIMARY cooperative level, no other application will be able to restore its buffers. Similarly, an application with the DSSCL_WRITEPRIMARY cooperative level must have the input focus to restore its primary sound buffer.

Once DirectSound restores the buffer memory, the application must rewrite the buffer with valid sound data. DirectSound cannot restore the contents of the memory, only the memory itself.

The application can receive notification that a buffer is lost when it specifies that buffer in a call to the **IDirectSoundBuffer::Lock** or **IDirectSoundBuffer::Play** method. These methods return DSERR_BUFFERLOST to indicate a lost buffer. The **IDirectSoundBuffer::GetStatus** method can also be used to retrieve the status of the sound buffer and test for the DSBSTATUS_BUFFERLOST flag.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::Lock**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::GetStatus**

IDirectSoundBuffer::SetCurrentPosition

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::SetCurrentPosition** method moves the current play position for secondary sound buffers.

```
HRESULT SetCurrentPosition(
  DWORD dwNewPosition
);
```

Parameters

dwNewPosition
New position, in bytes, from the beginning of the buffer that will be used when the sound buffer is played.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

This method cannot be called on primary sound buffers.

If the buffer is playing, it will immediately move to the new position and continue. If it is not playing, it will begin from the new position the next time the **IDirectSoundBuffer::Play** method is called.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Play**

IDirectSoundBuffer::SetFormat

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::SetFormat** method sets the format of the primary sound buffer for the application. Whenever this application has the input focus, DirectSound will set the primary buffer to the specified format.

```
HRESULT SetFormat(
  LPCWAVEFORMATEX lpcfxFormat
);
```

Parameters

lpcfxFormat
Address of a **WAVEFORMATEX** structure that describes the new format for the primary sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_BADFORMAT

DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY
DSERR_PRIOLEVELNEEDED
DSERR_UNSUPPORTED

Remarks

If this method is called on a primary buffer that is being accessed in write-primary cooperative level, the buffer must be stopped before **IDirectSoundBuffer::SetFormat** is called. If this method is being called on a primary buffer for a non-write-primary level, DirectSound will implicitly stop the primary buffer, change the format, and restart the primary; the application need not do this explicitly.

This method will succeed even if the hardware does not support the requested format. DirectSound will set the buffer to the closest supported format, then mix the sound in the requested format and convert it before sending it to the primary buffer. To determine whether this is happening, an application can call the **IDirectSoundBuffer::GetFormat** method for the primary buffer and compare the result with the format that was requested with the **SetFormat** method.

A call to this method also fails if the calling application has the DSSCL_NORMAL cooperative level.

This method is not valid for secondary sound buffers. If a secondary sound buffer requires a format change, the application should create a new DirectSoundBuffer object using the new format.

DirectSound supports PCM formats; it does not currently support compressed formats.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetFormat**

IDirectSoundBuffer::SetFrequency

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::SetFrequency** method sets the frequency at which the audio samples are played.

```
HRESULT SetFrequency(
  DWORD dwFrequency
);
```

Parameters

dwFrequency

New frequency, in hertz (Hz), at which to play the audio samples. The value must be in the range DSBFREQUENCY_MIN to DSBFREQUENCY_MAX. These values are currently defined in Dsound.h as 100 and 100,000 respectively.

If the value is DSBFREQUENCY_ORIGINAL, the frequency is reset to the default value in the current buffer format. This format is specified in the **IDirectSound::CreateSoundBuffer** method.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL

DSERR_GENERIC

DSERR_INVALIDPARAM

DSERR_PRIOLEVELNEEDED

Remarks

Increasing or decreasing the frequency changes the perceived pitch of the audio data. This method does not affect the format of the buffer.

This method is not valid for primary sound buffers.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSound::CreateSoundBuffer**, **IDirectSoundBuffer::GetFrequency**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::SetFormat**

IDirectSoundBuffer::SetPan

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::SetPan** method specifies the relative volume between the left and right channels.

```
HRESULT SetPan(
  LONG lPan
);
```

Parameters

lPan
: Relative volume between the left and right channels.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The value in *lPan* is measured in hundredths of a decibel (dB), in the range of DSBPAN_LEFT to DSBPAN_RIGHT. These values are currently defined in Dsound.h as -10,000 and 10,000 respectively. The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER, defined as zero. This value of 0 in the *lplPan* parameter means that both channels are at full volume (they are attenuated by 0 decibels). At any setting other than DSBPAN_CENTER, one of the channels is at full volume and the other is attenuated.

A pan of -2173 means that the left channel is at full volume and the right channel is attenuated by 21.73 dB. Similarly, a pan of 870 means that the left channel is attenuated by 8.7 dB and the right channel is at full volume. A pan of

DSBPAN_LEFT means that the right channel is silent and the sound is all the way to the left, while a pan of DSBPAN_RIGHT means that the left channel is silent and the sound is all the way to the right.

The pan control acts cumulatively with the volume control.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetPan**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::SetVolume

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::SetVolume** method changes the volume of a sound buffer.

```
HRESULT SetVolume(
  LONG lVolume
);
```

Parameters

lVolume
New volume requested for this sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The volume is specified in hundredths of decibels (dB). Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are currently defined in Dsound.h as 0 and -10,000 respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the stream. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for all practical purposes, is silence. Currently DirectSound does not support amplification.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetPan**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetPan**

IDirectSoundBuffer::Stop

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::Stop** method causes the sound buffer to stop playing.

```
HRESULT Stop();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

For secondary sound buffers, **IDirectSoundBuffer::Stop** will set the current position of the buffer to the sample that follows the last sample played. This means that if the **IDirectSoundBuffer::Play** method is called on the buffer, it will continue playing where it left off.

For primary sound buffers, if an application has the DSSCL_WRITEPRIMARY level, this method will stop the buffer and reset the current position to 0 (the beginning of the buffer). This is necessary because the primary buffers on most sound cards can play only from the beginning of the buffer.

However, if **IDirectSoundBuffer::Stop** is called on a primary buffer and the application has a cooperative level other than DSSCL_WRITEPRIMARY, this method simply reverses the effects of **IDirectSoundBuffer::Play**. It configures the primary buffer to stop if no secondary buffers are playing. If other buffers are playing in this or other applications, the primary buffer will not actually stop until they are stopped. This method is useful because playing the primary buffer consumes processing overhead even if the buffer is playing sound data with the amplitude of 0 decibels.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::Play**

IDirectSoundBuffer::Unlock

[This is preliminary documentation and subject to change.]

The **IDirectSoundBuffer::Unlock** method releases a locked sound buffer.

```
HRESULT Unlock(
  LPVOID lpvAudioPtr1,
  DWORD dwAudioBytes1,
  LPVOID lpvAudioPtr2,
  DWORD dwAudioBytes2
);
```

Parameters

lpvAudioPtr1
: Address of the value retrieved in the *lplpvAudioPtr1* parameter of the **IDirectSoundBuffer::Lock** method.

dwAudioBytes1
: Number of bytes actually written to the *lpvAudioPtr1* parameter. It should not exceed the number of bytes returned by the **IDirectSoundBuffer::Lock** method.

lpvAudioPtr2
: Address of the value retrieved in the *lplpvAudioPtr2* parameter of the **IDirectSoundBuffer::Lock** method.

dwAudioBytes2
: Number of bytes actually written to the *lpvAudioPtr2* parameter. It should not exceed the number of bytes returned by the **IDirectSoundBuffer::Lock** method.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

An application must pass both pointers, *lpvAudioPtr1* and *lpvAudioPtr2*, returned by the **IDirectSoundBuffer::Lock** method to ensure the correct pairing of **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Unlock**. The second pointer is needed even if 0 bytes were written to the second pointer.

Applications must pass the number of bytes actually written to the two pointers in the parameters *dwAudioBytes1* and *dwAudioBytes2*.

Make sure the sound buffer does not remain locked for long periods of time.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Lock**

IDirectSoundCapture

[This is preliminary documentation and subject to change.]

The methods of the **IDirectSoundCapture** interface are used to create sound capture buffers.

The interface is obtained by using the **DirectSoundCaptureCreate** function.

This reference section gives information on the following methods of the **IDirectSoundCapture** interface:

Creation	**CreateCaptureBuffer**
	Initialize
Capabilities	**GetCaps**

Like all COM interfaces, the **IDirectSoundCapture** interface also inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef

QueryInterface

Release

The **LPDIRECTSOUNDCAPTURE** type is defined as a pointer to the **IDirectSoundCapture** interface:

```
typedef struct IDirectSoundCapture    *LPDIRECTSOUNDCAPTURE;
```

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCapture::CreateCaptureBuffer

[This is preliminary documentation and subject to change.]

The **IDirectSoundCapture::CreateCaptureBuffer** method creates a capture buffer.

Unlike DirectSound, which can mix several sounds into one sound for output, DirectSoundCapture cannot do the exact opposite and extract various sounds from one input sound. For the first version, DirectSoundCapture allows only one capture buffer to exist at any given time per capture device.

```
HRESULT CreateCaptureBuffer(
  LPDSCBUFFERDESC lpDSCBufferDesc,
  LPLPDIRECTSOUNDCAPTUREBUFFER
lplpDirectSoundCaptureBuffer,
  LPUNKNOWN pUnkOuter
);
```

Parameters

lpDSCBufferDesc
: Pointer to a **DSCBUFFERDESC** structure containing values for the capture buffer being created.

lplpDirectSoundCaptureBuffer
: Address of the **IDirectSoundCaptureBuffer** interface pointer if successful.

pUnkOuter
: Controlling **IUnknown** of the aggregate. Its value must be NULL.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM

DSERR_BADFORMAT

DSERR_GENERIC

DSERR_NODRIVER

DSERR_OUTOFMEMORY

DSERR_UNINITIALIZED

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCapture::GetCaps

[This is preliminary documentation and subject to change.]

The **IDirectSoundCapture::GetCaps** method obtains the capabilities of the capture device.

```
HRESULT GetCaps(
  LPDSCCAPS lpDSCCaps
);
```

Parameters

lpDSCCaps
: Pointer to a **DSCCAPS** structure to be filled by the capture device. When the method is called, the **dwSize** member must specify the size of the structure in bytes.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM

DSERR_UNSUPPORTED

DSERR_NODRIVER

DSERR_OUTOFMEMORY

DSERR_UNINITIALIZED

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCapture::Initialize

[This is preliminary documentation and subject to change.]

When **CoCreateInstance** is used to create a DirectSoundCapture object, the object must be initialized with the **IDirectSoundCapture::Initialize** method. Calling this method is not required when the **DirectSoundCaptureCreate** function is used to create the object.

```
HRESULT Initialize(
  LPGUID lpGuid
);
```

Parameters

lpGuid
: Address of the GUID specifying the sound driver to which the DirectSoundCapture object binds. Use NULL to select the primary sound driver.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM

DSERR_NODRIVER

DSERR_OUTOFMEMORY

DSERR_ALREADYINITIALIZED

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCaptureBuffer

[This is preliminary documentation and subject to change.]

The methods of the **IDirectSoundCaptureBuffer** interface are used to manipulate sound capture buffers.

The interface is obtained by calling the **IDirectSoundCapture::CreateCaptureBuffer** method.

The methods of the **IDirectSoundCaptureBuffer** interface may be grouped as follows:

Initialization	**Initialize**
Information	**GetCaps**
	GetCurrentPosition
	GetFormat

	GetStatus
Capture management	**Lock**
	Start
	Stop
	Unlock

Like all COM interfaces, the **IDirectSoundCaptureBuffer** interface also inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef

QueryInterface

Release

The **LPDIRECTSOUNDCAPTUREBUFFER** type is defined as a pointer to the **IDirectSoundCaptureBuffer** interface:

```
typedef struct IDirectSoundCaptureBuffer  *LPDIRECTSOUNDCAPTUREBUFFER;
```

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCaptureBuffer::GetCaps

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::GetCaps** method returns the capabilities of the buffer.

```
HRESULT GetCaps(
  LPDSCBCAPS lpDSCBCaps
);
```

Parameters

lpDSCBCaps
: Pointer to a **DSCBCAPS** structure to be filled by the capture buffer. On input, the **dwSize** member must specify the size of the structure in bytes.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_UNSUPPORTED
DSERR_OUTOFMEMORY

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCaptureBuffer::GetCurrentPosition

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::GetCurrentPosition** method gets the current capture and read positions in the buffer.

The capture position is ahead of the read position. These positions are not always identical due to possible buffering of captured data either on the physical device or in the host. The data after the read position up to and including the capture position is not necessarily valid data.

```
HRESULT GetCurrentPosition(
  LPDWORD lpdwCapturePosition,
  LPDWORD lpdwReadPosition
);
```

Parameters

lpdwCapturePosition
: Address of a variable to receive the current capture position in the DirectSoundCaptureBuffer object. This position is an offset within the capture buffer and is specified in bytes. The value can be NULL if the caller is not interested in this position information.

lpdwReadPosition
: Address of a variable to receive the current position in the DirectSoundCaptureBuffer object at which it is safe to read data. This position is

an offset within the capture buffer and is specified in bytes. The value can be NULL if the caller is not interested in this position information.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_NODRIVER
DSERR_OUTOFMEMORY

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCaptureBuffer::GetFormat

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::GetFormat** method retrieves the current format of the capture buffer.

```
HRESULT GetFormat(
  LPWAVEFORMATEX lpwfxFormat,
  DWORD dwSizeAllocated,
  LPDWORD lpdwSizeWritten
);
```

Parameters

lpwfxFormat
Address of the **WAVEFORMATEX** variable to contain a description of the sound data in the capture buffer. To retrieve the buffer size needed to contain the format description, specify NULL. In this case the **DWORD** pointed to by the *lpdwSizeWritten* parameter will receive the size of the structure needed to receive complete format information.

dwSizeAllocated

Size, in bytes, of the **WAVEFORMATEX** structure. DirectSoundCapture writes, at most, **dwSizeAllocated** bytes to the structure; if the structure requires more memory, it is truncated.

lpdwSizeWritten
Address of a variable to receive the number of bytes written to the **WAVEFORMATEX** structure. This parameter can be NULL.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCaptureBuffer::GetStatus

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::GetStatus** method retrieves the current status of the capture buffer.

```
HRESULT GetStatus(
  DWORD *lpdwStatus
);
```

Parameters

lpdwStatus
Address of a variable to contain the status of the capture buffer. The status can be set to one or more of the following:

DSCBSTATUS_CAPTURING
DSCBSTATUS_LOOPING

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCaptureBuffer::Initialize

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::Initialize** method initializes a DirectSoundCaptureBuffer object if it has not yet been initialized.

```
HRESULT Initialize(
  LPDIRECTSOUNDCAPTURE lpDirectSoundCapture,
  LPCDSBUFFERDESC lpcDSBufferDesc
);
```

Parameters

lpDirectSoundCapture
Address of the DirectSoundCapture object associated with this DirectSoundCaptureBuffer object.

lpcDSBufferDesc
Address of a **DSBUFFERDESC** structure that contains the values used to initialize this sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_ALREADYINITIALIZED

Remarks

Because the **IDirectSoundCapture::CreateCaptureBuffer** method calls the **IDirectSoundCaptureBuffer::Initialize** method internally, it is not needed for the current release of DirectSound. This method is provided for future extensibility.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

DSBUFFERDESC, **IDirectSoundCapture::CreateCaptureBuffer**

IDirectSoundCaptureBuffer::Lock

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::Lock** method locks the buffer. Locking the buffer returns pointers into the buffer, allowing the application to read or write audio data into that memory.

```
HRESULT Lock(
  DWORD dwReadCursor,
  DWORD dwReadBytes,
  LPVOID *lplpvAudioPtr1,
  LPDWORD lpdwAudioBytes1,
  LPVOID *lplpvAudioPtr2,
  LPDWORD lpdwAudioBytes2,
  DWORD dwFlags
);
```

Parameters

dwReadCursor
Offset, in bytes, from the start of the buffer to where the lock begins.

dwReadBytes
Size, in bytes, of the portion of the buffer to lock. Note that the capture buffer is conceptually circular.

lplpvAudioPtr1
Address of a pointer to contain the first block of the capture buffer to be locked.

lpdwAudioBytes1
Address of a variable to contain the number of bytes pointed to by the *lplpvAudioPtr1* parameter. If this value is less than the *dwReadBytes* parameter, *lplpvAudioPtr2* will point to a second block of data.

lplpvAudioPtr2

Address of a pointer to contain the second block of the capture buffer to be locked. If the value of this parameter is NULL, the *lplpvAudioPtr1* parameter points to the entire locked portion of the capture buffer.

lpdwAudioBytes2

Address of a variable to contain the number of bytes pointed to by the *lplpvAudioPtr2* parameter. If *lplpvAudioPtr2* is NULL, this value will be 0.

dwFlags

Flags modifying the lock event. This value can be zero or the following flag:

DSCBLOCK_ENTIREBUFFER

The *dwReadBytes* parameter is to be ignored and the entire capture buffer is to be locked.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following values:

DSERR_INVALIDPARAM
DSERR_INVALIDCALL

Remarks

This method accepts an offset and a byte count, and returns two read pointers and their associated sizes. Two pointers are required because capture buffers are circular. If the locked bytes do not wrap around the end of the buffer, the second pointer, *lplpvAudioBytes2*, will be NULL. However, if the bytes do wrap around, then the second pointer will point to the beginning of the buffer.

If the application passes NULL for the *lplpvAudioPtr2* and *lpdwAudioBytes2* parameters, DirectSoundCapture will not lock the wraparound portion of the buffer.

The application should read data from the pointers returned by the **IDirectSoundCaptureBuffer::Lock** method and then call the **IDirectSoundCaptureBuffer::Unlock** method to release the buffer back to DirectSoundCapture. The sound buffer should not be locked for long periods of time; if it is, the capture cursor will reach the locked bytes and configuration-dependent audio problems may result.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCaptureBuffer::Start

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::Start** method puts the capture buffer into the capture state and begins capturing data into the buffer. If the capture buffer is already in the capture state then the method has no effect.

```
HRESULT Start(
  DWORD dwFlags
);
```

Parameters

dwFlags
: Flags that specify the behavior for the capture buffer when capturing sound data. Flags specifying how to play the buffer. The following flag is defined:

  DSCBSTART_LOOPING
  : Once the end of the buffer is reached, capture restarts at the beginning and continues until explicitly stopped.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM

DSERR_NODRIVER

DSERR_OUTOFMEMORY

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCaptureBuffer::Stop

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::Stop** method puts the capture buffer into the "stop" state and stops capturing data. If the capture buffer is already in the stop state then the method has no effect.

```
HRESULT Stop(
  void
);
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_NODRIVER
DSERR_OUTOFMEMORY

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundCaptureBuffer::Unlock

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::Unlock** method unlocks the buffer.

```
HRESULT Unlock(
  LPVOID lpvAudioPtr1,
  DWORD dwAudioBytes1,
  LPVOID lpvAudioPtr2,
  DWORD dwAudioBytes2
);
```

Parameters

lpvAudioPtr1
Address of the value retrieved in the *lplpvAudioPtr1* parameter of the **IDirectSoundCaptureBuffer::Lock** method.

dwAudioBytes1

Number of bytes actually read from the *lpvAudioPtr1* parameter. It should not exceed the number of bytes returned by the **IDirectSoundCaptureBuffer::Lock** method.

lpvAudioPtr2
Address of the value retrieved in the *lplpvAudioPtr2* parameter of the **IDirectSoundCaptureBuffer::Lock** method.

dwAudioBytes2
Number of bytes actually read from the *lpvAudioPtr2* parameter. It should not exceed the number of bytes returned by the **IDirectSoundCaptureBuffer::Lock** method.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following values:

DSERR_INVALIDPARAM
DSERR_INVALIDCALL

Remarks

An application must pass both pointers, *lpvAudioPtr1* and *lpvAudioPtr2*, returned by the **IDirectSoundCaptureBuffer::Lock** method to ensure the correct pairing of **IDirectSoundCaptureBuffer::Lock** and **IDirectSoundCaptureBuffer::Unlock**. The second pointer is needed even if zero bytes were written to the second pointer.

Applications must pass the number of bytes actually written to the two pointers in the parameters *dwAudioBytes1* and *dwAudioBytes2*.

Make sure that the capture buffer does not remain locked for long periods of time.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundNotify

[This is preliminary documentation and subject to change.]

The **IDirectSoundNotify** interface provides a mechanism for setting up notification events for a playback or capture buffer.

The interface is obtained by calling the **QueryInterface** method of an existing interface on a DirectSoundBuffer object. For an example, see Play Buffer Notification.

The interface has the following method:

SetNotificationPositions

Like all COM interfaces, the **IDirectSoundNotify** interface also inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef

QueryInterface

Release

The **LPDIRECTSOUNDNOTIFY** type is defined as a pointer to the **IDirectSoundNotify** interface:

```
typedef struct IDirectSoundNotify    *LPDIRECTSOUNDNOTIFY;
```

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

IDirectSoundNotify::SetNotificationPositions

[This is preliminary documentation and subject to change.]

The **IDirectSoundCaptureBuffer::SetNotificationPositions** method sets the notification positions. During capture or playback, whenever the position reaches an offset specified in one of the **DSBPOSITIONNOTIFY** structures in the caller-supplied array, the associated event will be signaled. The position tracked in playback is the current play position; in capture it is the current read position.

```
HRESULT SetNotificationPositions(
  DWORD cPositionNotifies,
  LPCDSBPOSITIONNOTIFY lpcPositionNotifies
);
```

Parameters

cPositionNotifies
Number of **DSBPOSITIONNOTIFY** structures.

lpcPositionNotifies
Pointer to an array of **DSBPOSITIONNOTIFY** structures.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY

Remarks

The value DSBPN_OFFSETSTOP can be specified in the **dwOffset** member to tell DirectSound to signal the associated event when the **IDirectSoundBuffer::Stop or IDirectSoundCaptureBuffer::Stop** method is called or when the end of the buffer has been reached and the playback is not looping. If it is used, this should be the last item in the position-notify array.

If a position-notify array has already been set, calling this function again will replace the previous position-notify array.

The buffer must be stopped when this method is called.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

Functions

[This is preliminary documentation and subject to change.]

This section contains reference information for the following DirectSound and DirectSoundCapture global functions:

- **DirectSoundCaptureCreate**
- **DirectSoundCaptureEnumerate**
- **DirectSoundCreate**

- **DirectSoundEnumerate**

DirectSoundCaptureCreate

[This is preliminary documentation and subject to change.]

The **DirectSoundCaptureCreate** function creates and initializes an object that supports the **IDirectSoundCapture** interface.

HRESULT WINAPI DirectSoundCaptureCreate(
  LPGUID *lpGUID*,
  LPDIRECTSOUNDCAPTURE **lplpDSC*,
  LPUNKNOWN *pUnkOuter*
);

Parameters

lpGUID
  Address of the GUID that identifies the sound capture device. The value of this parameter must be one of the GUIDs returned by **DirectSoundCaptureEnumerate**, or NULL for the default device.

lplpDSC
  Address of a pointer to a DirectSoundCapture object created in response to this function.

pUnkOuter
  Controlling unknown of the aggregate. Its value must be NULL.

Return Values

If the function succeeds, the return value is DS_OK.

If the function fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM

DSERR_NOAGGREGATION

DSERR_OUTOFMEMORY

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

DirectSoundCaptureEnumerate

[This is preliminary documentation and subject to change.]

The **DirectSoundCaptureEnumerate** function enumerates the DirectSoundCapture objects installed in the system.

```
HRESULT WINAPI DirectSoundCaptureEnumerate(
  LPDSENUMCALLBACK lpDSEnumCallback,
  LPVOID lpContext
);
```

Parameters

lpDSEnumCallback
: Address of the **DSEnumCallback** function that will be called for each DirectSoundCapture object installed in the system.

lpContext
: Address of the user-defined context passed to the enumeration callback function every time that function is called.

Return Values

If the function succeeds, the return value is DS_OK.

If the function fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

DirectSoundCreate

[This is preliminary documentation and subject to change.]

The **DirectSoundCreate** function creates and initializes an **IDirectSound** interface.

```
HRESULT WINAPI DirectSoundCreate(
  LPGUID lpGuid,
  LPDIRECTSOUND * ppDS,
  LPUNKNOWN * pUnkOuter
```

```
);
```

Parameters

lpGuid
Address of the GUID that identifies the sound device. The value of this parameter must be one of the GUIDs returned by **DirectSoundEnumerate**, or NULL for the default device.

ppDS
Address of a pointer to a DirectSound object created in response to this function.

pUnkOuter
Controlling unknown of the aggregate. Its value must be NULL.

Return Values

If the function succeeds, the return value is DS_OK.

If the function fails, the return value may be one of the following error values:

DSERR_ALLOCATED
DSERR_INVALIDPARAM
DSERR_NOAGGREGATION
DSERR_NODRIVER
DSERR_OUTOFMEMORY

Remarks

The application must call the **IDirectSound::SetCooperativeLevel** method immediately after creating a DirectSound object.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

IDirectSound::GetCaps, **IDirectSound::SetCooperativeLevel**

DirectSoundEnumerate

[This is preliminary documentation and subject to change.]

The **DirectSoundEnumerate** function enumerates the DirectSound drivers installed in the system.

```
HRESULT WINAPI DirectSoundEnumerate(
  LPDSENUMCALLBACK lpDSEnumCallback,
  LPVOID lpContext
);
```

Parameters

lpDSEnumCallback
: Address of the **DSEnumCallback** function that will be called for each DirectSound object installed in the system.

lpContext
: Address of the user-defined context passed to the enumeration callback function every time that function is called.

Return Values

If the function succeeds, the return value is DS_OK.

If the function fails, the return value may be DSERR_INVALIDPARAM.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: Use dsound.lib.

See Also

DSEnumCallback

Callback Function

[This is preliminary documentation and subject to change.]

This section contains reference information for the following DirectSound callback function.

- **DSEnumCallback**

DSEnumCallback

[This is preliminary documentation and subject to change.]

DSEnumCallback is an application-defined callback function that enumerates the DirectSound drivers. The system calls this function in response to the application's previous call to the **DirectSoundEnumerate** or **DirectSoundCaptureEnumerate** function.

```
BOOL CALLBACK DSEnumCallback(
  LPGUID lpGuid,
  LPCSTR lpcstrDescription,
  LPCSTR lpcstrModule,
  LPVOID lpContext
);
```

Parameters

lpGuid
: Address of the GUID that identifies the DirectSound driver being enumerated. This value can be passed to the **DirectSoundCreate** function to create a DirectSound object for that driver.

lpcstrDescription
: Address of a null-terminated string that provides a textual description of the DirectSound device.

lpcstrModule
: Address of a null-terminated string that specifies the module name of the DirectSound driver corresponding to this device.

lpContext
: Address of application-defined data that is passed to each callback function.

Return Values

Returns TRUE to continue enumerating drivers, or FALSE to stop.

Remarks

The application can save the strings passed in the *lpcstrDescription* and *lpcstrModule* parameters by copying them to memory allocated from the heap. The memory used to pass the strings to this callback function is valid only while this callback function is running.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.
Import Library: User-defined.

See Also

DirectSoundEnumerate

Structures

[This is preliminary documentation and subject to change.]

This section contains reference information for the following structures used with DirectSound:

- **DS3DBUFFER**
- **DS3DLISTENER**
- **DSBCAPS**
- **DSBPOSITIONNOTIFY**
- **DSBUFFERDESC**
- **DSCAPS**
- **DSCBCAPS**
- **DSCBUFFERDESC**
- **DSCCAPS**

DS3DBUFFER

[This is preliminary documentation and subject to change.]

The **DS3DBUFFER** structure contains all information necessary to uniquely describe the location, orientation, and motion of a 3-D sound buffer. This structure is used with the **IDirectSound3DBuffer::GetAllParameters** and **IDirectSound3DBuffer::SetAllParameters** methods.

```
typedef struct {
    DWORD       dwSize;
    D3DVECTOR  vPosition;
    D3DVECTOR  vVelocity;
    DWORD       dwInsideConeAngle;
    DWORD       dwOutsideConeAngle;
    D3DVECTOR  vConeOrientation;
```

```
    LONG       lConeOutsideVolume;
    D3DVALUE   flMinDistance;
    D3DVALUE   flMaxDistance;
    DWORD      dwMode;
} DS3DBUFFER, *LPDS3DBUFFER;

typedef const DS3DBUFFER *LPCDS3DBUFFER;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

vPosition
A **D3DVECTOR** structure that describes the current position of the 3-D sound buffer.

vVelocity
A **D3DVECTOR** structure that describes the current velocity of the 3-D sound buffer.

dwInsideConeAngle
The angle of the inside sound projection cone.

dwOutsideConeAngle
The angle of the outside sound projection cone.

vConeOrientation
A **D3DVECTOR** structure that describes the current orientation of this 3-D buffer's sound projection cone.

lConeOutsideVolume
The cone outside volume.

flMinDistance
The minimum distance.

flMaxDistance
The maximum distance.

dwMode
The 3-D sound processing mode to be set.

DS3DMODE_DISABLE
3-D sound processing is disabled. The sound will appear to originate from the center of the listener's head.

DS3DMODE_HEADRELATIVE
Sound parameters (position, velocity, and orientation) are relative to the listener's parameters. In this mode, the absolute parameters of the sound are updated automatically as the listener's parameters change, so that the relative parameters remain constant.

DS3DMODE_NORMAL

Normal processing. This is the default mode.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.

DS3DLISTENER

[This is preliminary documentation and subject to change.]

The **DS3DLISTENER** structure contains all information necessary to uniquely describe the 3-D world parameters and position of the listener. This structure is used with the **IDirectSound3DListener::GetAllParameters** and **IDirectSound3DListener::SetAllParameters** methods.

```
typedef struct {
    DWORD      dwSize;
    D3DVECTOR  vPosition;
    D3DVECTOR  vVelocity;
    D3DVECTOR  vOrientFront;
    D3DVECTOR  vOrientTop;
    D3DVALUE   flDistanceFactor;
    D3DVALUE   flRolloffFactor;
    D3DVALUE   flDopplerFactor;
} DS3DLISTENER, *LPDS3DLISTENER;

typedef const DS3DLISTENER *LPCDS3DLISTENER;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

vPosition, **vVelocity**, **vOrientFront**, and **vOrientTop**
D3DVECTOR structures that describe the listener's position, velocity, front orientation, and top orientation, respectively.

flDistanceFactor, **flRolloffFactor**, and **flDopplerFactor**
The current distance, rolloff, and Doppler factors, respectively.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.

DSBCAPS

[This is preliminary documentation and subject to change.]

The **DSBCAPS** structure specifies the capabilities of a DirectSound buffer object, for use by the **IDirectSoundBuffer::GetCaps** method.

```
typedef struct {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwBufferBytes;
    DWORD dwUnlockTransferRate;
    DWORD dwPlayCpuOverhead;
} DSBCAPS, *LPDSBCAPS;

typedef const DSBCAPS *LPCDSBCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Flags that specify buffer-object capabilities.

DSBCAPS_CTRL3D
The buffer is either a primary buffer or a secondary buffer that uses 3-D control. To create a primary buffer, the **dwFlags** member of the **DSBUFFERDESC** structure should include the DSBCAPS_PRIMARYBUFFER flag.

DSBCAPS_CTRLFREQUENCY
The buffer must have frequency control capability.

DSBCAPS_CTRLPAN
The buffer must have pan control capability.

DSBCAPS_CTRLVOLUME
The buffer must have volume control capability.

DSBCAPS_GETCURRENTPOSITION2
Indicates that **IDirectSoundBuffer::GetCurrentPosition** should use the new behavior of the play cursor. In DirectSound in DirectX 1, the play cursor was

significantly ahead of the actual playing sound on emulated sound cards; it was directly behind the write cursor. Now, if the DSBCAPS_GETCURRENTPOSITION2 flag is specified, the application can get a more accurate play position. If this flag is not specified, the old behavior is preserved for compatibility. Note that this flag affects only emulated sound cards; if a DirectSound driver is present, the play cursor is accurate for DirectSound in all versions of DirectX.

DSBCAPS_GLOBALFOCUS
The buffer is a global sound buffer. With this flag set, an application using DirectSound can continue to play its buffers if the user switches focus to another application, even if the new application uses DirectSound. The one exception is if you switch focus to a DirectSound application that uses the DSSCL_EXCLUSIVE or DSSCL_WRITEPRIMARY flag for its cooperative level. In this case, the global sounds from other applications will not be audible.

DSBCAPS_LOCHARDWARE
The buffer is in hardware memory and uses hardware mixing.

DSBCAPS_LOCSOFTWARE
The buffer is in software memory and uses software mixing.

DSBCAPS_MUTE3DATMAXDISTANCE
The sound is reduced to silence at the maximum distance. The buffer will stop playing when the maximum distance is exceeded, so that processor time is not wasted.

DSBCAPS_PRIMARYBUFFER
Indicates that the buffer is a primary sound buffer. If this value is not specified, a secondary sound buffer will be created.

DSBCAPS_STATIC
Indicates that the buffer will be used for static sound data. Typically, these buffers are loaded once and played many times. These buffers are candidates for hardware memory.

DSBCAPS_STICKYFOCUS
Changes the focus behavior of the sound buffer. This flag can be specified in an **IDirectSound::CreateSoundBuffer** call. With this flag set, an application using DirectSound can continue to play its sticky focus buffers if the user switches to another application not using DirectSound. In this situation, the application's normal buffers are muted, but the sticky focus buffers are still audible. This is useful for nongame applications, such as movie playback (DirectShow™), when the user wants to hear the soundtrack while typing in Microsoft Word or Microsoft® Excel, for example. However, if the user switches to another DirectSound application, all sound buffers, both normal and sticky focus, in the previous application are muted.

dwBufferBytes
Size of this buffer, in bytes.

dwUnlockTransferRate

Specifies the rate, in kilobytes per second, at which data is transferred to the buffer memory when **IDirectSoundBuffer::Unlock** is called. High-performance applications can use this value to determine the time required for **IDirectSoundBuffer::Unlock** to execute. For software buffers located in system memory, the rate will be very high because no processing is required. For hardware buffers, the rate might be slower because the buffer might have to be downloaded to the sound card, which might have a limited transfer rate.

dwPlayCpuOverhead

Specifies the processing overhead as a percentage of main processing cycles needed to mix this sound buffer. For hardware buffers, this member will be 0 because the mixing is performed by the sound device. For software buffers, this member depends on the buffer format and the speed of the system processor.

Remarks

The **DSBCAPS** structure contains information similar to that found in the **DSBUFFERDESC** structure passed to the **IDirectSound::CreateSoundBuffer** method, with some additional information. Additional information includes the location of the buffer (hardware or software) and some cost measures (such as the time to download the buffer if located in hardware, and the processing overhead to play the buffer if it is mixed in software).

Note

The **dwFlags** member of the **DSBCAPS** structure contains the same flags used by the **DSBUFFERDESC** structure. The only difference is that in the **DSBCAPS** structure, either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag will be specified, according to the location of the buffer memory. In the **DSBUFFERDESC** structure, these flags are optional and are used to force the buffer to be located in either hardware or software.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.

See Also

IDirectSound::CreateSoundBuffer, **IDirectSoundBuffer::GetCaps**

DSBPOSITIONNOTIFY

[This is preliminary documentation and subject to change.]

The **DSBPOSITIONNOTIFY** structure is used by the **IDirectSoundNotify::SetNotificationPositions** method.

```
typedef struct {
    DWORD   dwOffset;
    HANDLE  hEventNotify;
} DSBPOSITIONNOTIFY, *LPDSBPOSITIONNOTIFY;

typedef const DSBPOSITIONNOTIFY *LPCDSBPOSITIONNOTIFY;
```

Members

dwOffset
: Offset from the beginning of the buffer where the notify event is to be triggered, or DSBPN_OFFSETSTOP.

hEventNotify
: Handle to the event to be signaled when the offset has been reached.

Remarks

The DSBPN_OFFSETSTOP value in the **dwOffset** member causes the event to be signaled when playback or capture stops, either because the end of the buffer has been reached (and playback or capture is not looping) or because the application called the **IDirectSoundBuffer::Stop** or **IDirectSoundCaptureBuffer::Stop** method.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.

DSBUFFERDESC

[This is preliminary documentation and subject to change.]

The **DSBUFFERDESC** structure describes the necessary characteristics of a new DirectSoundBuffer object. This structure is used by the **IDirectSound::CreateSoundBuffer** method.

```
typedef struct {
    DWORD         dwSize;
    DWORD         dwFlags;
    DWORD         dwBufferBytes;
    DWORD         dwReserved;
```

```
    LPWAVEFORMATEX  lpwfxFormat;
} DSBUFFERDESC, *LPDSBUFFERDESC;

typedef const DSBUFFERDESC *LPCDSBUFFERDESC;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Identifies the capabilities to include when creating a new DirectSoundBuffer object. Specify one or more of the following:

DSBCAPS_CTRL3D
The buffer is either a primary buffer or a secondary buffer that uses 3-D control.

DSBCAPS_CTRLALL
The buffer must have all control capabilities.

DSBCAPS_CTRLDEFAULT
The buffer should have default control options. This is the same as specifying the DSBCAPS_CTRLPAN, DSBCAPS_CTRLVOLUME, and DSBCAPS_CTRLFREQUENCY flags.

DSBCAPS_CTRLFREQUENCY
The buffer must have frequency control capability.

DSBCAPS_CTRLPAN
The buffer must have pan control capability.

DSBCAPS_CTRLPOSITIONNOTIFY
The buffer must have position notification capability.

DSBCAPS_CTRLVOLUME
The buffer must have volume control capability.

DSBCAPS_GETCURRENTPOSITION2
Indicates that **IDirectSoundBuffer::GetCurrentPosition** should use the new behavior of the play cursor. In DirectSound in DirectX 1, the play cursor was significantly ahead of the actual playing sound on emulated sound cards; it was directly behind the write cursor. Now, if the DSBCAPS_GETCURRENTPOSITION2 flag is specified, the application can get a more accurate play position. If this flag is not specified, the old behavior is preserved for compatibility. Note that this flag affects only emulated sound cards; if a DirectSound driver is present, the play cursor is accurate for DirectSound in all versions of DirectX.

DSBCAPS_GLOBALFOCUS
The buffer is a global sound buffer. With this flag set, an application using DirectSound can continue to play its buffers if the user switches focus to another application, even if the new application uses DirectSound. The one

exception is if you switch focus to a DirectSound application that uses the DSSCL_EXCLUSIVE or DSSCL_WRITEPRIMARY flag for its cooperative level. In this case, the global sounds from other applications will not be audible.

DSBCAPS_LOCHARDWARE

Forces the buffer to use hardware mixing, even if DSBCAPS_STATIC is not specified. If the device does not support hardware mixing or if the required hardware memory is not available, the call to the **IDirectSound::CreateSoundBuffer** method will fail. The application must ensure that a mixing channel will be available for this buffer; this condition is not guaranteed.

DSBCAPS_LOCSOFTWARE

Forces the buffer to be stored in software memory and use software mixing, even if DSBCAPS_STATIC is specified and hardware resources are available.

DSBCAPS_MUTE3DATMAXDISTANCE

The sound is to be reduced to silence at the maximum distance. The buffer will stop playing when the maximum distance is exceeded, so that processor time is not wasted.

DSBCAPS_PRIMARYBUFFER

Indicates that the buffer is a primary sound buffer. If this value is not specified, a secondary sound buffer will be created.

DSBCAPS_STATIC

Indicates that the buffer will be used for static sound data. Typically, these buffers are loaded once and played many times. These buffers are candidates for hardware memory.

DSBCAPS_STICKYFOCUS

Changes the focus behavior of the sound buffer. This flag can be specified in an **IDirectSound::CreateSoundBuffer** call. With this flag set, an application using DirectSound can continue to play its sticky focus buffers if the user switches to another application not using DirectSound. In this situation, the application's normal buffers are muted, but the sticky focus buffers are still audible. This is useful for nongame applications, such as movie playback (DirectShow), when the user wants to hear the soundtrack while typing in Microsoft Word or Microsoft Excel, for example. However, if the user switches to another DirectSound application, all sound buffers, both normal and sticky focus, in the previous application are muted.

dwBufferBytes

Size of the new buffer, in bytes. This value must be 0 when creating primary buffers. For secondary buffers, the minimum and maximum sizes allowed are specified by DSBSIZE_MIN and DSBSIZE_MAX, defined in Dsound.h.

dwReserved

This value is reserved. Do not use.

lpwfxFormat

Address of a structure specifying the waveform format for the buffer. This value must be NULL for primary buffers. The application can use **IDirectSoundBuffer::SetFormat** to set the format of the primary buffer.

Remarks

The DSBCAPS_LOCHARDWARE and DSBCAPS_LOCSOFTWARE flags used in the **dwFlags** member are optional and mutually exclusive. DSBCAPS_LOCHARDWARE forces the buffer to reside in memory located in the sound card. DSBCAPS_LOCSOFTWARE forces the buffer to reside in main system memory, if possible.

These flags are also defined for the **dwFlags** member of the **DSBCAPS** structure, and when used there, the specified flag indicates the actual location of the DirectSoundBuffer object.

When creating a primary buffer, applications must set the **dwBufferBytes** member to 0; DirectSound will determine the optimal buffer size for the particular sound device in use. To determine the size of a created primary buffer, call **IDirectSoundBuffer::GetCaps**.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.

See Also

IDirectSound::CreateSoundBuffer

DSCAPS

[This is preliminary documentation and subject to change.]

The **DSCAPS** structure specifies the capabilities of a DirectSound device for use by the **IDirectSound::GetCaps** method.

```
typedef {
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwMinSecondarySampleRate;
    DWORD  dwMaxSecondarySampleRate;
    DWORD  dwPrimaryBuffers;
    DWORD  dwMaxHwMixingAllBuffers;
    DWORD  dwMaxHwMixingStaticBuffers;
```

```
    DWORD  dwMaxHwMixingStreamingBuffers;
    DWORD  dwFreeHwMixingAllBuffers;
    DWORD  dwFreeHwMixingStaticBuffers;
    DWORD  dwFreeHwMixingStreamingBuffers;
    DWORD  dwMaxHw3DAllBuffers;
    DWORD  dwMaxHw3DStaticBuffers;
    DWORD  dwMaxHw3DStreamingBuffers;
    DWORD  dwFreeHw3DAllBuffers;
    DWORD  dwFreeHw3DStaticBuffers;
    DWORD  dwFreeHw3DStreamingBuffers;
    DWORD  dwTotalHwMemBytes;
    DWORD  dwFreeHwMemBytes;
    DWORD  dwMaxContigFreeHwMemBytes;
    DWORD  dwUnlockTransferRateHwBuffers;
    DWORD  dwPlayCpuOverheadSwBuffers;
    DWORD  dwReserved1;
    DWORD  dwReserved2;
} DSCAPS, *LPDSCAPS;

typedef const DSCAPS *LPCDSCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Specifies device capabilities. Can be one or more of the following:

DSCAPS_CERTIFIED
This driver has been tested and certified by Microsoft.

DSCAPS_CONTINUOUSRATE
The device supports all sample rates between the **dwMinSecondarySampleRate** and **dwMaxSecondarySampleRate** member values. Typically, this means that the actual output rate will be within +/- 10 hertz (Hz) of the requested frequency.

DSCAPS_EMULDRIVER
The device does not have a DirectSound driver installed, so it is being emulated through the waveform-audio functions. Performance degradation should be expected.

DSCAPS_PRIMARY16BIT
The device supports primary sound buffers with 16-bit samples.

DSCAPS_PRIMARY8BIT
The device supports primary buffers with 8-bit samples.

DSCAPS_PRIMARYMONO

The device supports monophonic primary buffers.

DSCAPS_PRIMARYSTEREO
The device supports stereo primary buffers.

DSCAPS_SECONDARY16BIT
The device supports hardware-mixed secondary sound buffers with 16-bit samples.

DSCAPS_SECONDARY8BIT
The device supports hardware-mixed secondary buffers with 8-bit samples.

DSCAPS_SECONDARYMONO
The device supports hardware-mixed monophonic secondary buffers.

DSCAPS_SECONDARYSTEREO
The device supports hardware-mixed stereo secondary buffers.

dwMinSecondarySampleRate and **dwMaxSecondarySampleRate**
Minimum and maximum sample rate specifications that are supported by this device's hardware secondary sound buffers.

dwPrimaryBuffers
Number of primary buffers supported. This value will always be 1.

dwMaxHwMixingAllBuffers
Specifies the total number of buffers that can be mixed in hardware. This member can be less than the sum of **dwMaxHwMixingStaticBuffers** and **dwMaxHwMixingStreamingBuffers**. Resource tradeoffs frequently occur.

dwMaxHwMixingStaticBuffers
Specifies the maximum number of static sound buffers.

dwMaxHwMixingStreamingBuffers
Specifies the maximum number of streaming sound buffers.

dwFreeHwMixingAllBuffers, **dwFreeHwMixingStaticBuffers**, and **dwFreeHwMixingStreamingBuffers**
Description of the free, or unallocated, hardware mixing capabilities of the device. An application can use these values to determine whether hardware resources are available for allocation to a secondary sound buffer. Also, by comparing these values to the members that specify maximum mixing capabilities, the resources that are already allocated can be determined.

dwMaxHw3DAllBuffers, **dwMaxHw3DStaticBuffers**, and **dwMaxHw3DStreamingBuffers**
Description of the hardware 3-D positional capabilities of the device.

dwFreeHw3DAllBuffers, **dwFreeHw3DStaticBuffers**, and **dwFreeHw3DStreamingBuffers**
Description of the free, or unallocated, hardware 3-D positional capabilities of the device.

dwTotalHwMemBytes
Size, in bytes, of the amount of memory on the sound card that stores static sound buffers.

dwFreeHwMemBytes
Size, in bytes, of the free memory on the sound card.

dwMaxContigFreeHwMemBytes
Size, in bytes, of the largest contiguous block of free memory on the sound card.

dwUnlockTransferRateHwBuffers
Description of the rate, in kilobytes per second, at which data can be transferred to hardware static sound buffers. This and the number of bytes transferred determines the duration of a call to the **IDirectSoundBuffer::Unlock** method.

dwPlayCpuOverheadSwBuffers
Description of the processing overhead, as a percentage of the central processing unit, needed to mix software buffers (those located in main system memory). This varies according to the bus type, the processor type, and the clock speed.

The unlock transfer rate for software buffers is 0 because the data need not be transferred anywhere. Similarly, the play processing overhead for hardware buffers is 0 because the mixing is done by the sound device.

dwReserved1 and **dwReserved2**
Reserved for future use.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.

See Also

IDirectSound::GetCaps

DSCBCAPS

[This is preliminary documentation and subject to change.]

The **DSCBCAPS** structure is used by the **IDirectSoundCaptureBuffer::GetCaps** method.

```
typedef struct
{
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwBufferBytes;
    DWORD   dwReserved;
} DSCBCAPS, *LPDSCBCAPS;

typedef const DSCBCAPS *LPCDSCBCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Specifies device capabilities. Can be zero or the following flag:

DSCBCAPS_WAVEMAPPED
The Win32 wave mapper will be used for formats not supported by the device.

dwBufferBytes
The size, in bytes, of the capture buffer.

dwReserved
Reserved for future use.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.

DSCBUFFERDESC

[This is preliminary documentation and subject to change.]

The **DSCBUFFERDESC** structure is used by the **IDirectSoundCapture::CreateCaptureBuffer** method.

```
typedef struct
{
    DWORD           dwSize;
    DWORD           dwFlags;
    DWORD           dwBufferBytes;
    DWORD           dwReserved;
    LPWAVEFORMATEX      lpwfxFormat;
} DSCBUFFERDESC, *LPDSCBUFFERDESC;

typedef const DSCBUFFERDESC *LPCDSCBUFFERDESC;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Specifies device capabilities. Can be zero or the following flag:

DSCBCAPS_WAVEMAPPED
The Win32 wave mapper will be used for formats not supported by the device.

dwBufferBytes
Size of capture buffer to create, in bytes.

dwReserved
Reserved for future use.

lpwfxFormat
Pointer to a **WAVEFORMATEX** structure containing the format in which to capture the data.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.

DSCCAPS

[This is preliminary documentation and subject to change.]

The **DSCCAPS** structure is used by the **IDirectSoundCapture::GetCaps** method.

```
typedef struct
{
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwFormats;
    DWORD   dwChannels;
} DSCCAPS, *LPDSCCAPS;

typedef const DSCCAPS *LPCDSCCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Specifies device capabilities. Can be zero or one of the following flags:

DSCCAPS_EMULDRIVER

There is no DirectSoundCapture driver for the device, so the standard wave audio functions are being used.

DSCCAPS_CERTIFIED

The WDM driver is certified.

dwFormats

Standard formats that are supported. These are equivalent to the values in the **WAVEINCAPS** structure used in the Win32 waveform audio functions, and are reproduced here for convenience.

Value	Meaning
WAVE_FORMAT_1M08	11.025 kHz, mono, 8-bit
WAVE_FORMAT_1M16	11.025 kHz, mono, 16-bit
WAVE_FORMAT_1S08	11.025 kHz, stereo, 8-bit
WAVE_FORMAT_1S16	11.025 kHz, stereo, 16-bit
WAVE_FORMAT_2M08	22.05 kHz, mono, 8-bit
WAVE_FORMAT_2M16	22.05 kHz, mono, 16-bit
WAVE_FORMAT_2S08	22.05 kHz, stereo, 8-bit
WAVE_FORMAT_2S16	22.05 kHz, stereo, 16-bit
WAVE_FORMAT_4M08	44.1 kHz, mono, 8-bit
WAVE_FORMAT_4M16	44.1 kHz, mono, 16-bit
WAVE_FORMAT_4S08	44.1 kHz, stereo, 8-bit
WAVE_FORMAT_4S16	44.1 kHz, stereo, 16-bit

dwChannels

Number specifying the number of channels supported by the device, where 1 is mono, 2 is stereo, and so on.

QuickInfo

Windows NT: Requires version 4.0 SP3 or later.
Windows: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Header: Declared in dsound.h.

Return Values

[This is preliminary documentation and subject to change.]

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all **IDirectSound** and **IDirectSoundBuffer** methods. For a list of the error codes each method can return, see the individual method descriptions.

DS_OK
The request completed successfully.

DSERR_ALLOCATED
The request failed because resources, such as a priority level, were already in use by another caller.

DSERR_ALREADYINITIALIZED
The object is already initialized.

DSERR_BADFORMAT
The specified wave format is not supported.

DSERR_BUFFERLOST
The buffer memory has been lost and must be restored.

DSERR_CONTROLUNAVAIL
The control (volume, pan, and so forth) requested by the caller is not available.

DSERR_GENERIC
An undetermined error occurred inside the DirectSound subsystem.

DSERR_INVALIDCALL
This function is not valid for the current state of this object.

DSERR_INVALIDPARAM
An invalid parameter was passed to the returning function.

DSERR_NOAGGREGATION
The object does not support aggregation.

DSERR_NODRIVER
No sound driver is available for use.

DSERR_NOINTERFACE
The requested COM interface is not available.

DSERR_OTHERAPPHASPRIO
Another application has a higher priority level, preventing this call from succeeding

DSERR_OUTOFMEMORY
The DirectSound subsystem could not allocate sufficient memory to complete the caller's request.

DSERR_PRIOLEVELNEEDED
The caller does not have the priority level required for the function to succeed.

DSERR_UNINITIALIZED
The **IDirectSound::Initialize** method has not been called or has not been called successfully before other methods were called.

DSERR_UNSUPPORTED
The function called is not supported at this time.

DirectSound Samples

[This is preliminary documentation and subject to change.]

This section provides summaries of the applications in the DirectX SDK that are primarily intended to demonstrate the DirectSound component. The following sample programs demonstrate the use and capabilities of DirectSound:

- DS3DView
- DSShow
- DSShow3D
- DSStream
- FDFilter

DS3DView

[This is preliminary documentation and subject to change.]

Description

The DS3DView program is an extension of the Direct3D Viewer application. In addition to displaying an object in 3-D, DS3DView enables you to attach a sounds to the object. The object can then be moved. As it moves, its sound changes in the way that the sounds of real moving objects change. That is, DirectSound simulates attenuation and the Doppler effect.

Required Components

DirectX 2 or later.

Path

Source: \Mssdk\Samples\Multimedia\DSound\src\DS3DView

Executable: \Mssdk\Samples\Multimedia\DSound\bin

User's Guide

Run the program and choose **File** from the main menu. Select **Add 3D Visual** from the **File** menu. You can find X files containing the definitions of 3-D objects in the directory dxsdk\sdk\media. When you select one of the sample objects, DS3DViewer will display it on the screen.

Attach a sound to the object by selecting **Sound** from the main menu. When the **Sound** menu is displayed, choose **Attach Sound**. The DirectX SDK ships with a variety of sample sound Wav files. You'll find them in the directory dxdsk\dsk\media.

After you select a sound, you can hear it once by choosing **Play Sound Once** from the **Sound** menu. To hear it continuously, select **Play Sound Looping** from the **Sound** menu. To hear some of DirectSound's effects, try selecting **Orbit Selected Object** from the **Motion** menu. This option will orbit the object around the listener and add the appropriate orbital sound effects. You can also select **Bullet Selected Object**, which will shoot the object past the listener's head with the accompanying sound effects.

Programming Notes

In addition to 3-D retained mode techniques, this program illustrates how to load Wav files, attach them to objects in a 3-D environment, and move them with realistic sound effects.

DSShow

[This is preliminary documentation and subject to change.]

Description

This example application demonstrates many of the capabilities of DirectSound.

Required Components

DirectX 5.0 or later.

Path

Source: \Mssdk\Samples\Multimedia\DSound\src\DSShow

Executable: \Mssdk\Samples\Multimedia\DSound\bin

User's Guide

With the DSShow application, you can open one or more sound files and examine many DirectSound features. For each Wav file that you open, DSShow displays file information and a group of controls in the client area of its window. Use the controls to alter the way that DirectSound plays the sound. You can pan the sound to the right or left, adjust the volume, and adjust the frequency. The sound can be played once by selecting the **Play** button. If you want the sound to play continuously, check the **Looped** checkbox. To close the sound file, select the **Close** button.

You can alter the output frequency, and select whether the sound is played in stereo or mono by choosing **Output Type** from the **Options** menu.

The **Check Latency** item in the **Options** menu enables you to test how rapidly a sound starts and stops after it is triggered.

In addition, you can tell the DSShow application to enumerate all available sound drivers whenever it starts up by selecting **Enumerate Drivers** from the **Options** menu.

Programming Notes

The DSShow title bar displays information about the hardware mixing capabilities of your sound card. It shows the number of free mixing channels and the available memory. If both of those numbers are 0, your card does not have hardware mixing.

DSShow3D

[This is preliminary documentation and subject to change.]

Description

This example application demonstrates many of the 3-D sound capabilities of DirectSound. It is an extension of the DSShow sample application.

Required Components

DirectX 5.0 or later.

Path

Source: \Mssdk\Samples\Multimedia\DSound\src\DSShow3D

Executable: \Mssdk\Samples\Multimedia\DSound\bin

User's Guide

As with the DSShow application, you can open one or more sound files and play them at the same time. For each Wav file that you open, DSShow3D displays a group of controls in a client window. Use the controls to alter the way that DirectSound plays the sound. You can pan the sound to the right or left, adjust the volume, and adjust the frequency. The sound can be played once by selecting the **Play** button. If you want the sound to play continuously, select the **Looped** button. To close the sound file, choose **Close** from the client window's **File** menu.

You can alter the output frequency, and select whether the sound is played in stereo or mono by choosing **Output Format** from the **Options** menu.

To demonstrate the 3-D sound capabilities of DirectSound, select **Settings** from the **Options** menu. Select the checkbox labeled **Open 3D as Default**. Any sound files that are subsequently opened will be opened as 3-D sound files. The DSShow3D program will display slider controls for altering the 3-D characteristics of the sound.

Programming Notes

This program features a compact interface, support for 3-D sounds and new buffer options, the ability to open as many sounds as your hardware allows, and a new user interface that takes advantage of the entire desktop for placing (or minimizing) windows.

DSStream

[This is preliminary documentation and subject to change.]

Description

The DSStream sample illustrates the use of streaming sound buffers. It works best with sounds that are more than 2 seconds in length.

Required Components

DirectX 5.0 or later.

Path

Source: \Mssdk\Samples\Multimedia\DSound\src\DSStream

Executable: \Mssdk\Samples\Multimedia\DSound\bin

User's Guide

Load a Wav file by selecting **Open** from the **File** menu. Play the sound once by choosing **Play!** from the main menu or by selecting the **Play** button. Halt the playback by choosing **Stop!** from the main menu or by selecting the **Stop** button.

To make the program play the sound continuously, select the checkbox labeled **Loope** (yes, that is the way it's spelled in the program).

The DSStream program displays controls for panning the sound, decreasing the volume, and changing the frequency.

As with DSShow, you can make DSStream enumerate the sound drivers when it starts up. To do this, select **Enumerate Drivers** from the **Options** menu. You must then exit the program and restart it.

Programming Notes

The DSStream program is a variation on the DSShow example, butdoes not contain as many features. This program can only open and stream sound from one file at a time. However, that is a limitation of this program, not of DirectSound.

FDFilter

[This is preliminary documentation and subject to change.]

Description

The FDFilter program demonstrates how to use DirectSound to implement a full duplex audio and a filter. In this case, the input is filtered to add a gargling sound to it.

Required Components

DirectX 5.0 or later.

Speakers. Sound card. Microphone.

Path

Source: \Mssdk\Samples\Multimedia\DSound\src\FDfilter

Executable: \Mssdk\Samples\Multimedia\DSound\bin

User's Guide

When you start this program, it will present you with a dialog box that contains pull-down lists of the sound input and output devices attached to your computer. Select the appropriate devices. Usually the defaults are the most appropriate. It will next display lists of sampling rates for the input and output sounds. Select the values that are appropriate for your sound card. The program will then show a dialog box that enables you to select the filter. The available options are **None** (pass-through), which just plays back the input sound as is, or **Gargle**, which distorts the input sound into a gargling noise.