DirectSound

This section provides information about the DirectSound® component of DirectX®. Information is found under the following main headings:

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

About DirectSound

The Microsoft® DirectSound® application programming interface (API) is the audio component of the DirectX® Programmer's Reference of the Platform Software Development Kit (SDK). 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.

New in this release of DirectSound are capabilities for audio capture. Also new is support for property sets, which enable application developers to take advantage of extended services offered by sound cards and their associated drivers.

Why Use DirectSound?

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
- Developing 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 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
- Playback Overview
- Capture Overview
- Property Sets Overview
- Hardware Abstraction and Emulation
- System Integration

Architectural Overview

DirectSound implements a new model for playing and capturing digital sound samples and mixing sample sources. As with other object classes in 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.

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

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

Playback Overview

The DirectSound buffer object represents a buffer containing sound data in pulse code modulation (PCM) format. 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.

DirectSound does not include functions for parsing a sound file. It is your responsibility to stream data in the correct format into the secondary sound buffers.

Normally, buffers from only a single application are audible at any given moment, because only one application at a time has access to a particular DirectSound device.

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.

Through the **IDirectSoundNotify** interface, DirectSound provides a mechanism to notify the client when the play cursor reaches positions within a buffer that have been specified by the client, or when playback has stopped.

Capture Overview

The **DirectSoundCapture** object is used to query the capabilities of sound capture devices and to create buffers for capturing audio from an input source.

Audio capture functions already exist in Win32. The first release of **DirectSoundCapture** in the DirectX 5 Programmer's Reference does not provide any performance improvement over the existing **waveIn** functions. However, the **DirectSoundCapture** API allows application developers to create titles using consistent interfaces for both audio playback and capture. It also allows titles to be developed today that will benefit from new, improved driver models and API implementations in the future.

DirectSoundCapture allows capturing of compressed formats. In this first version, the underlying **waveIn** functions or the hardware provide support for compressed formats. **DirectSoundCapture** does not call the audio compression manager (ACM) functions itself.

The DirectSoundCaptureBuffer object represents a buffer used for capturing data from the input device. This buffer is circular; that is, when the input pointer reaches the end of the buffer, it 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

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

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.

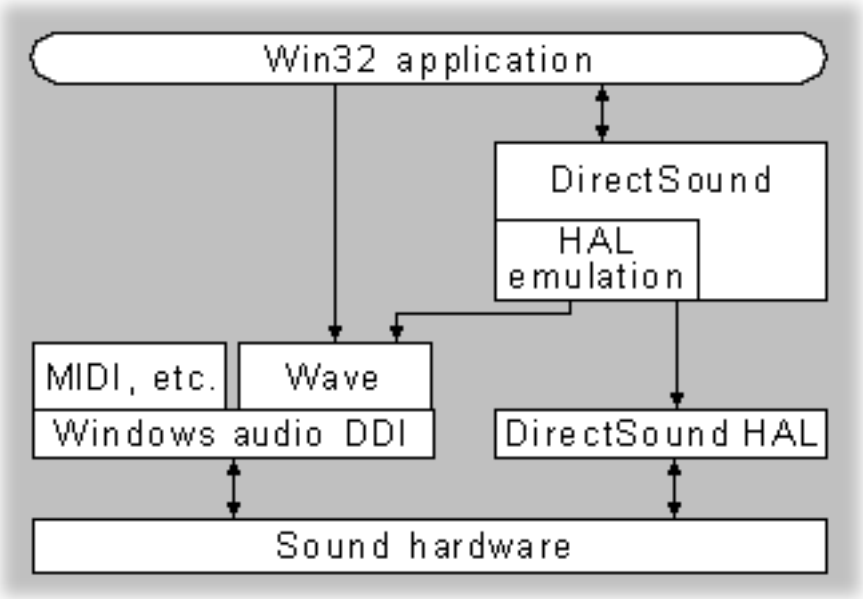
Your application can use DirectSound as long as the DirectX run-time files are present on the user's system. If the sound hardware does not have an installed DirectSound driver, DirectSound uses its hardware-emulation layer (HEL), which employs the Windows multimedia waveform-audio (**waveIn** and **waveOut**)

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

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 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
- Introduction to 3-D Sound
- DirectSound 3-D Buffers
- DirectSound 3-D Listeners
- DirectSoundCapture
- DirectSound Property Sets
- Optimizing DirectSound Performance

Note

Most of the examples of method calls throughout this section are given in the C 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);
```

You can simplify C calls to any of the DirectSound methods by using the macros defined in Dsound.h. For example:

```
IDirectSound_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);
```

DirectSound Devices

The first step in implementing DirectSound in an application is to create a DirectSound object, which creates an instance of the **IDirectSound** interface.

A DirectSound object describes the audio hardware on a system. The **IDirectSound** interface enables your application to define and control the sound card, speaker, and memory environment.

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

For an application that is simply going to play sounds through the user’s preferred playback device, you need not 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 will fail.

However, if you are looking for a particular kind of device 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 GUID for each device
- Allows you to initialize DirectSound for each device as it is enumerated

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).

If you are working with more than one device—for example, a capture and a playback device—the callback function is a good place to create and initialize the DirectSound object for each device.

The following example, extracted from Dsenum.c in the Dsshow sample, enumerates the available devices and adds information about each to a list in a combo box. Here is the callback function in its entirety:

```
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 (DirectSoundEnumerate((LPDSENUMCALLBACK)DSEnumProc, &hCombo)
    != DS_OK )
    {
    EndDialog( hDlg, TRUE );
    return( TRUE );
    }
```

Note that the address of the handle to the combo box 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

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 lpDirectSound;
HRESULT       hr;
hr = DirectSoundCreate(NULL, &lpDirectSound, NULL));
```

The function returns an error if there is no sound device or if the sound device is being used by 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.

```
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 driver GUID parameter that **DirectSoundCreate** typically 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, shut down COM by calling the **CoUninitialize** function, as follows:

```
CoUninitialize();
```

Cooperative Levels

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 normal cooperative level, which allows for orderly switching between applications that use the sound card.

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 going on.

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

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.

It is unwise to make assumptions about the behavior of the sound device; if you do, your application might work on some sound devices but not on others. Furthermore, future devices might behave differently.

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

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

Compacting Hardware Memory

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 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

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, provided the original buffer is not on the sound hardware.

You can mix sounds from different secondary buffers simply by playing them at the same time. Data from secondary buffers is mixed by DirectSound in the primary buffer. Any number of secondary buffers can be played at one time, up to the limits of 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

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

To create a sound buffer, your application fills a **DSBUFFERDESC** structure and then calls 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(DS_OK == hr) {
        // Succeeded. 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

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. DirectSound uses these options when it allocates hardware resources 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.

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

If your application calls a method that a buffer lacks, that method fails. For example, if you attempt to change the volume by using the **IDirectSoundBuffer::SetVolume** method, the method succeeds if the DSBCAPS_CTRLVOLUME flag was specified when the buffer was created. Otherwise the method fails and returns the DSERR_CONTROLUNAVAIL error code. Providing controls for the buffers helps to ensure that all applications run correctly on all existing or future sound devices.

Access to the Primary Buffer

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.

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;
    // 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(&lplpDsb, 0, sizeof(DSBUFFERDESC)); // Zero it out.
    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 (DS_OK == hr) {
       // Succeeded. Try to create buffer.
       hr = lpDirectSound->lpVtbl->CreateSoundBuffer(lpDirectSound,
          &dsbdesc, lplpDsb, NULL);
       if (DS_OK == hr) {
          // Succeeded. Set primary buffer to desired format.
          hr = (*lplpDsb)->lpVtbl->SetFormat(*lplpDsb, &pcmwf);
          if (DS_OK == 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;
          }
       }
    }
    // SetCooperativeLevel failed.
    // CreateSoundBuffer, or SetFormat.
    *lplpDsb = NULL;
    *lpdwBufferSize = 0;
    return FALSE;
}
```

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.

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

Playing a sound consists of the following steps:

1. Lock a portion of the secondary buffer (**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.
2. Write the audio data to the buffer.
3. Unlock the buffer (**IDirectSoundBuffer::Unlock**).
4. Send the sound to the primary buffer and from there to the output device (**IDirectSoundBuffer::Play**).

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 300 bytes beginning at the midpoint of a 400-byte buffer, the **Lock** method would return one pointer to the last 200 bytes of the buffer, and a second pointer to the first 100 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 will play repeatedly until the application calls the **IDirectSoundBuffer::Stop** method, at which point the play cursor is moved to the beginning of the buffer.

For streaming sound buffers, your application is responsible for ensuring that the next block of data is written to the buffer before the current play position loops back to the beginning. (For more on the play position, see Current Play and Write Positions.) You can do this by setting notification positions so that an event is signaled whenever the current play position reaches a certain point. 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(DS_OK == 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(DS_OK == hr) {
            // Success.
            return TRUE;
        }
    }
    // Lock, Unlock, or Restore failed.
    return FALSE;
}
```

Playback Controls

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.

Current Play and Write Positions

DirectSound maintains two pointers into the buffer: the current play position (or 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 where it is.

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.

Play Buffer Notification

Particularly when streaming audio, you may want your application to be notified when the play cursor 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 with 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,
                                &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

It is easy to mix multiple streams with DirectSound. You simply create secondary sound buffers, receiving 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

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 obtain 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** 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 (DS_OK == 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 (DS_OK == hr) {
            // Success.
            return TRUE;
        }
    }
    // Lock or Unlock failed.
    return FALSE;
}
```

Buffer Management

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. In particular, this can occur when buffers are located in the hardware sound memory. In the worst case, the sound card itself might be removed from the system while in use; this situation can occur with PCMCIA sound cards.

Loss can also occur when an application with the write-primary cooperative level moves to the foreground. If this flag is set, 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 any other buffer. When the application 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 does not contain valid sound data. The owning application must rewrite the data to the restored buffer.

Compressed Wave Formats

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) data 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.

Introduction to 3-D Sound

DirectSound enables an application to change the apparent position and movement of a sound source. A sound source can be a point from which sounds radiate in all directions or a cone outside which sounds are attenuated. Applications can also modify sounds using Doppler shift.

Although these effects are audible using standard loudspeakers, they are more obvious and compelling when the user wears headphones.

This overview introduces the basic concepts of 3-D sound as implemented by DirectSound. The following topics are discussed:

- Perception of Sound Positions
- Listeners
- Sound Cones
- Distance Measurements
- Doppler Shift
- Integration with Direct3D
- Mono and Stereo Sources

Specific 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

Perception of Sound Positions

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.
- *Arrival offset.* 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 sounds 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. When hardware that supports 3-D sound becomes generally available, other positioning cues might be incorporated into the system, including the difference in how high- and low-frequency sounds are muffled by the mass of the listener's head and the reflections of sound off the shoulders and earlobes.

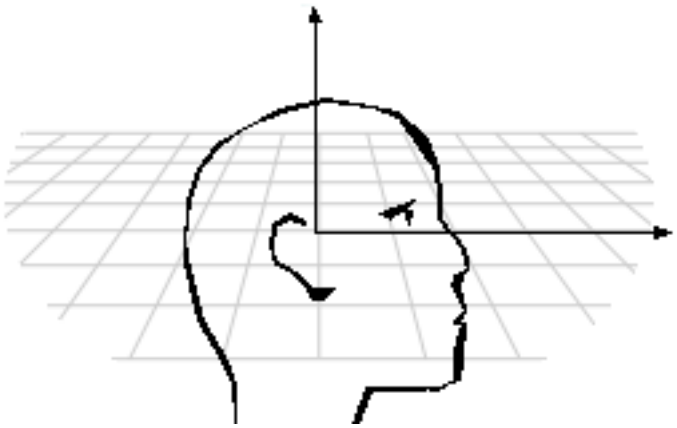
One of the most important sound-positioning cues is the apparent visual position of the sound source. If a projectile appears as a dot in the distance and grows to the size

of an intercontinental missile before it roars past the viewer's head, the listener does not need subtle acoustical cues in order to perceive that the sound has gone past.

Listeners

Listeners experience an identical sonic effect when an object moves in a 90-degree arc around them or if they move their heads 90 degrees relative to the object. Programmatically, however, it is often much simpler to change the position or orientation of the listener than to change the position of every other object in a scene. DirectSound makes this possible through the **IDirectSound3DListener** interface.

Listener *orientation* is defined by the relationship between two vectors that share an origin: the *top* and *front vectors*. The top vector originates from the center of the listener's head and points straight up through the top of the head. The front vector also originates from the center of the listener's head, but it points at a right angle to the top vector, forward through the listener's face. The following illustration shows the directions of these vectors:

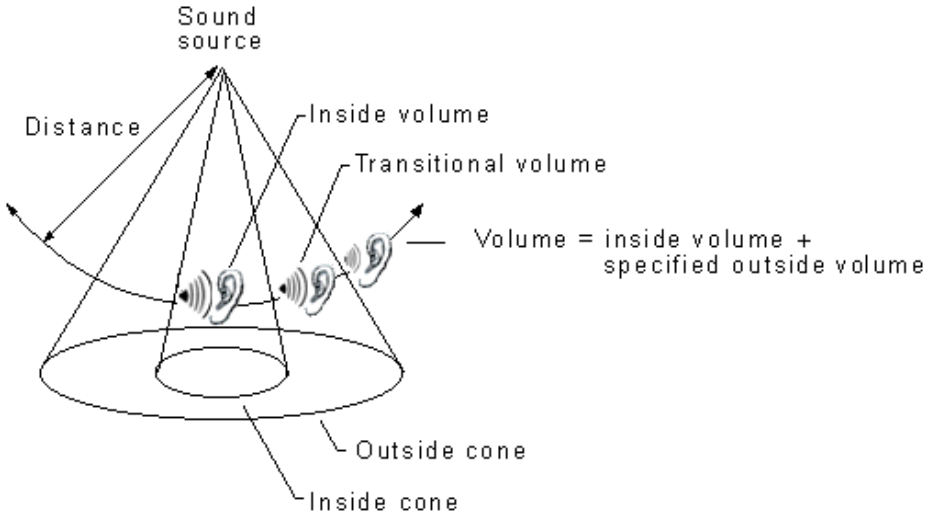


Sound Cones

A sound with a position but no orientation is a point source; the farther the listener is from the sound, in any direction, the quieter the sound. A sound with a position and an orientation is a sound cone.

In DirectSound, sound cones include an inside cone and an outside cone. Within the inside cone, the volume is at the maximum level for that sound source. (Because DirectSound does not support amplification, the maximum volume level is zero; all other volume levels are negative values that represent an attenuation of the maximum volume.) Outside the outside cone, the volume is the specified outside volume added to the inside volume. If an application sets the outside volume to DSBVOLUME_MIN, for example, the sound source will be inaudible outside the outside cone. Between the outside and inside cones, the volume changes gradually from one level to the other.

The concept of sound cones is shown in the following illustration:



Technically, every sound buffer represented by the **IDirectSound3DBuffer** interface is a sound cone, but often these sound cones behave like omnidirectional sound sources. For example, the default value for the volume outside the sound cone is zero; unless the application changes this value, the volume will be the same inside and outside the cone, and sound will not have any apparent orientation. You could also make the sound-cone angles as wide as you want, effectively making the sound cone a sphere.

Designing sound cones properly can add dramatic effects to your application. For example, you could position the sound source in the center of a room, setting its orientation toward a door. Then set the angle of the outside cone so that it extends to the width of the doorway and set the outside cone volume to inaudible. The user, when passing the open door, will suddenly hear the voice emanating from the room.

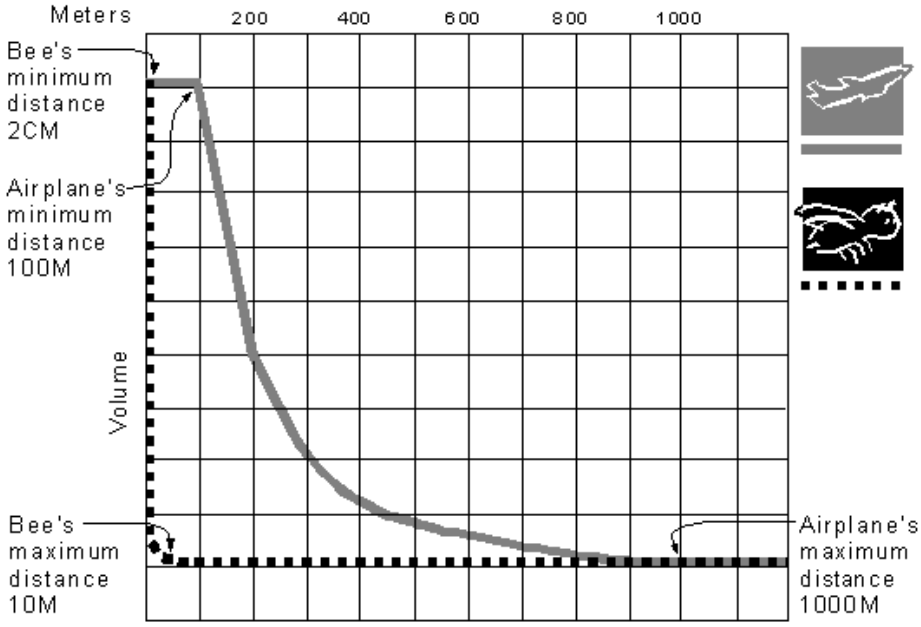
Distance Measurements

The 3-D effects of DirectSound use meters as the default unit of distance measurements. If your application does not use meters, it need not convert between units of measure to maintain compatibility with the component. Instead, the application 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 as its unit of measure, it could specify a distance factor of .30480006096, which is the number of meters in a foot.

The default distance measurements for the 3-D sound effects mimic the natural world. Many application designers choose to change these values, however, to make the effects more dramatic. Exaggerated Doppler effects or exaggerated sound attenuation with distance can make an application more exciting.

As a listener approaches a sound source, the sound gets louder. Past a certain point, however, it is not reasonable for the volume to continue to increase; either the maximum (zero) has been reached, or the nature of the sound source imposes a logical limit. This is the *minimum distance* for the sound source. Similarly, the *maximum distance* for a sound source is the distance beyond which the sound does not get any quieter.

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 (16-bit audio doesn't have enough room to accommodate such different volume levels). 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. This concept is shown in the following illustration:



Doppler Shift

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

The velocity of a sound source or listener does not necessarily reflect the speed at which it is moving through space. Setting an object’s velocity does not move it, nor does moving the object affect the velocity. Velocity is simply a vector used to

calculate the Doppler shift. In order to have realistic Doppler shift effects in your application, you must calculate the velocity of any object that is moving and set the appropriate value for that sound source or listener. You are free to exaggerate or minimize this value in order to create special effects.

You can also globally increase or decrease Doppler shift effects by setting the *Doppler factor* for the listener.

Integration with Direct3D

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** function to monitor the movement of a frame in an application and change the sonic environment only when a certain condition has been reached.

Mono and Stereo Sources

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.

DirectSound 3-D Buffers

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 successfully 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
- Operation Mode
- Buffer Position and Velocity
- Cone Parameters

Obtaining the IDirectSound3DBuffer Interface

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,
                                &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

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

Applications can specify the distances at which 3-D sounds stop getting louder or quieter. For an overview of these values, see Distance Measurements.

The default minimum distance, defined in Dsound.h as DS3D_DEFAULTMINDISTANCE, is currently 1 distance unit (normally 1 meter). The default maximum distance, defined as DS3D_DEFAULTMAXDISTANCE, is effectively infinite.

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.

Operation Mode

Sound buffers have three processing modes: normal, head-relative, and disabled. Normal processing is the default mode. In the head-relative mode, 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. In the 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. This method sets the operation mode based on the flag the application sets for the first parameter, *dwMode*.

Buffer Position and Velocity

An application can set and retrieve a sound source’s position in 3-D space by using the **IDirectSound3DBuffer::SetPosition** and **IDirectSound3DBuffer::GetPosition** methods.

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. A buffer's position is not affected by its velocity. Velocity is measured in distance units per second – by default, meters per second.

See also:

- Doppler Shift
- Distance Measurements

Cone Parameters

An application sets or retrieves the angles that define sound cones by using the **IDirectSound3DBuffer::SetConeAngles** and **IDirectSound3DBuffer::GetConeAngles** methods. To set or retrieve the orientation of sound cones, an application can use the **IDirectSound3DBuffer::SetConeOrientation** and **IDirectSound3DBuffer::GetConeOrientation** methods.

By default, cone angles are 360 degrees, meaning the object projects sound at the same volume in all directions. A smaller value means that the object projects sound at a lower volume outside the defined angle. The outside cone angle must always be equal to or greater than the inside cone angle.

The outside cone volume represents the additional volume attenuation of the sound when the listener is outside the buffer's sound cone. This factor is expressed in hundredths of decibels. By default the outside volume is zero, meaning the sound cone will have no perceptible effect.

An application sets and retrieves the outside cone volume by using the **IDirectSound3DBuffer::SetConeOutsideVolume** and **IDirectSound3DBuffer::GetConeOutsideVolume** methods. Keep in mind that an audible outside cone volume is still subject to attenuation, due to distance from the sound source.

When the listener is within the sound cone, the normal buffer volume (returned by the **IDirectSoundBuffer::GetVolume** method) is used.

For a conceptual overview, see Sound Cones.

DirectSound 3-D Listeners

A 3-D listener represents the person who hears sounds generated by sound buffer objects in 3-D space. The **IDirectSound3DListener** interface controls the listener's position and apparent velocity in 3-D space. It also controls the environment parameters that affect the behavior of the DirectSound component, such as the amount of Doppler shifting and volume attenuation applied to sound sources far from the listener.

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
- Doppler Factor
- Listener Position and Velocity
- Listener Orientation

- Rolloff Factor

Obtaining the IDirectSound3DListener Interface

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:

```
// LPDIRECTSOUNDBUFFER lpDsbPrimary;
// The buffer has been created with DSBCAPS_CTRL3D.
LPDIRECTSOUND3DLISTENER lpDs3dListener;

HRESULT hr = lpDsbPrimary->QueryInterface(IID_IDirectSound3DListener,
                                &lpDs3dListener);

if (SUCCEEDED(hr))
 {
 // Perform 3-D operations.
 .
 .
 .
 }
```

Batch Parameters for IDirectSound3DListener

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

Changes to 3-D sound buffer and listener settings such as position, velocity, and Doppler factor will cause the DirectSound mixer to remix its mix-ahead buffer, 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 with a single remix of the mix-ahead buffer.

Note

Any deferred settings are overwritten if your application calls the same setting method with the DS3D_IMMEDIATE flag before it calls **IDirectSound3DListener::CommitDeferredSettings**. For example, if you set the listener velocity to (1,2,3) with the deferred flag and then set the listener velocity to (4,5,6) with the immediate flag, the velocity will be (4,5,6). Then, if your application calls the **IDirectSound3DListener::CommitDeferredSettings** method, the velocity will still be (4,5,6).

Distance Factor

DirectSound uses meters as the default unit of distance measurements. If your application does not use meters, it can set a distance factor. For an overview, see Distance Measurements.

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 with the **IDirectSound3DListener::GetDistanceFactor** method. The default value is DS3D_DEFAULTDISTANCEFACTOR, defined as 1.0, meaning that a distance unit corresponds to 1 meter. At the default value, a position vector of (3.0,7.2,-20.9) means that the object is 3.0 m to the right of, 7.2 m above, and 20.9 m behind the origin. If the distance factor is changed to 2.0, the same position vector means that the object is 6.0 m to the right of, 14.4 m above, and 41.8 m behind the origin.

Doppler Factor

DirectSound applies Doppler-shift effects to sounds, based on the relative velocity of the listener and the sound buffer. (For an overview, see Doppler Shift.) The Doppler shift can be ignored, exaggerated, or given the same effect as in the real world, depending on a variable called 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 two times the real-world Doppler shift; and so on.

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

Listener Position and Velocity

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. A listener's position is not affected by its velocity.

Listener Orientation

The listener's orientation plays a strong role in 3-D effects processing. DirectSound approximates sound cues to provide the illusion that a sound is generated at a particular point in space. For more information about these cues, see Perception of Sound Positions.

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).

For an illustration of the front and top vectors, see Listeners.

Rolloff Factor

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 two times the real-world rolloff, and so on.

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

DirectSoundCapture

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 present version of DirectSound does not permit capturing and mixing from multiple devices at the same time.) The actual capturing is done with the methods of the buffer object.

This section covers the following topics:

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

Creating the DirectSoundCapture Object

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

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

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

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. These 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

Creating a Capture Buffer

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. See the reference for **WAVEFORMATEX** in the Win32 API section of the Platform SDK for information on the members of that structure.

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

```
DSCBUFFERDESC           dscbd;
LPDIRECTSOUNDCAPTUREBUFFER pDSCB;
WAVEFORMATEX            wfx =
{
   // wFormatTag, nChannels, nSamplesPerSec, mAvgBytesPerSec,
   // nBlockAlign, wBitsPerSample, cbSize
   {WAVE_FORMAT_PCM, 2, 44100, 176400, 4, 16, 0},
};
dscbd.dwSize = sizeof(DSCBUFFERDESC);
dscbd.dwFlags = 0;
// We're going to capture one second's worth of audio
dscbd.dwBufferBytes = wfx.nAvgBytesPerSec;
dscbd.dwReserved = 0;
dscbd.lpwfxFormat = &wfx;

pDSCB = NULL;

// pDSC is the pointer to the DirectSoundCapture object
```

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

Capture Buffer Information

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. See the reference for **WAVEFORMATEX** in the Win32 API section of the Platform SDK for information on the members of that structure.

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

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 of an 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. 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

//  LPDIRECTSOUNDNOTIFY  lpDsNotify;
//    lpDsNotify was initialized with QueryInterface.
//  WAVEFORMATEX   wfx;
//    wfx was initialized when the buffer was created.
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;
   }
}

// Set notification positions.
// Notify us when read position is halfway through the buffer,
// assuming buffer holds one second of audio.
rgdsbpn[0].dwOffset = (wfx.nAvgBytesPerSec/2) -1;
rgdsbpn[0].hEventNotify = rghEvent[0];
// Notify us when capture is at the end of the buffer.
rgdsbpn[1].dwOffset = wfx.nAvgBytesPerSec - 1;
rgdsbpn[1].hEventNotify = rghEvent[1];
rgdsbpn[2].dwOffset = DSBPN_OFFSETSTOP;
rgdsbpn[2].hEventNotify = rghEvent[2];

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

Capturing Sounds

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. See Capture Buffer Notification for one method of determining when the capture position reaches a certain point.
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 should 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 all the data is captured. 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, at which point the capture position is moved to the beginning of the buffer.

DirectSound Property Sets

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_ReverbProperties** containing properties such as **DSPROPERTY_REVERBPROPERTIES_HALL** and **DSPROPERTY_REVERBPROPERTIES_STADIUM**.

Typically, the property identifiers are defined using a C language enumeration starting at ordinal 0.

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::QuerySupport** 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 famous arias in the voices of several tenors. The driver developer creates a property set, **DSPROPSETID_Aria**, containing properties like **DSPROPERTY_ARIA_VESTI_LA_GIUBBA** and **DSPROPERTY_ARIA_CHE_GELIDA_MANINA**. 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 the great aria from *Pagliacci*.

```
DWORD WhichTenor = CARUSO;
BOOL  StartOrStop = START;

HRESULT hr = lpKsPropertySet->Set(
            DSPROPSETID_Aria,
            KSPROPERTY_ARIA_VESTI_LA_GIUBBA,
            &WhichTenor,
            sizeof(WhichTenor),
```

```
&StartOrStop),
sizeof(StartOrStop);
```

Optimizing DirectSound Performance

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

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

Matching Buffer Formats

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 (that is, wave files) and primary buffer have the same format.

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).

Reducing DMA Overhead

Most of today's sound cards are ISA-bus cards that use DMA (direct memory access) to move sound data from system memory to local buffers. This DMA activity directly affects CPU performance when the processor is forced to wait for a DMA transfer to end before it can access memory. This performance hit is unavoidable on ISA sound cards but is not a problem with the newer PCI cards.

DMA overhead could be the biggest single factor affecting the performance of DirectSound. Fortunately, this factor is easy to control when you're tweaking performance.

The impact of DMA overhead is directly related to the data rate of the primary buffer. Experiment with reducing the data rate requirement by changing the format of the primary buffer. For more information on how to do this, see Access to the Primary Buffer and **IDirectSoundBuffer::SetFormat**.

Playing the Primary Buffer Continuously

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

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

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, especially if you're sending them in streams or in bursts. 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 once instead of twice per frame.

Note
3-D control changes (orientation, position, velocity, Doppler factor, and so on) also cause the DirectSound mixer 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

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.

DirectSound Reference

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 section contains references for methods of the following DirectSound interfaces:

- **IDirectSound**
- **IDirectSound3DBuffer**
- **IDirectSound3DListener**

- **IDirectSoundCapture**
- **IDirectSoundCaptureBuffer**
- **IDirectSoundNotify**
- **IKsPropertySet**

IDirectSound

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:

Allocating memory	**Compact**
	Initialize
Creating buffers	**CreateSoundBuffer**
	DuplicateSoundBuffer
	SetCooperativeLevel
Device capabilities	**GetCaps**
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;
```

IDirectSound::Compact

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

If the application calls this method, it must have exclusive cooperation with the DirectSound object. (To get exclusive access, specify DSSCL_EXCLUSIVE in a call to the **IDirectSound::SetCooperativeLevel** method.) This method will fail if any operations are in progress.

IDirectSound::CreateSoundBuffer

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.

See Also

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

IDirectSound::DuplicateSoundBuffer

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.

See Also

IDirectSound::CreateSoundBuffer

IDirectSound::GetCaps

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.

See Also

DirectSoundCreate, **DSCAPS**

IDirectSound::GetSpeakerConfig

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)
    {...}
  }
```

See Also

IDirectSound::SetSpeakerConfig

IDirectSound::Initialize

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 for this DirectSound object to bind to. 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.

See Also

DirectSoundCreate

IDirectSound::SetCooperativeLevel

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. Most applications should use this level, because it has the smoothest multitasking and resource-sharing behavior.

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_NORMAL; use other priority levels when necessary. For additional information, see Cooperative Levels.

See Also

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

IDirectSound::SetSpeakerConfig

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));
```

See Also

IDirectSound::GetSpeakerConfig

IDirectSound3DBuffer

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;
```

IDirectSound3DBuffer::GetAllParameters

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.

IDirectSound3DBuffer::GetConeAngles

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.

IDirectSound3DBuffer::GetConeOrientation

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

This method has no effect unless the cone angle and cone volume factor have also been set. The default value is (0,0,1).

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 at right angles and have a magnitude of <= 1.0.

See Also

IDirectSound3DBuffer::SetConeOrientation,

IDirectSound3DBuffer::SetConeAngles,
IDirectSound3DBuffer::SetConeOutsideVolume

IDirectSound3DBuffer::GetConeOutsideVolume

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.

See Also

IDirectSoundBuffer::SetVolume

IDirectSound3DBuffer::GetMaxDistance

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.

See Also

IDirectSound3DBuffer::GetMinDistance,
IDirectSound3DBuffer::SetMaxDistance

IDirectSound3DBuffer::GetMinDistance

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 m per unit).

See Also

IDirectSound3DBuffer::SetMinDistance,
IDirectSound3DBuffer::GetMaxDistance

IDirectSound3DBuffer::GetMode

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.

IDirectSound3DBuffer::GetPosition

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.

IDirectSound3DBuffer::GetVelocity

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 Shift.

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

See Also

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

IDirectSound3DBuffer::SetAllParameters

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.

IDirectSound3DBuffer::SetConeAngles

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.

See Also

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

IDirectSound3DBuffer::SetConeOrientation

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.

Remarks

The vector information in the *lpvOrientation* parameter of the **IDirectSound3DBuffer::GetConeOrientation** method represents the center of the sound cone. The default value is (0,0,1).

See Also

IDirectSound3DBuffer::SetConeAngles,
IDirectSound3DBuffer::SetConeOutsideVolume

IDirectSound3DBuffer::SetConeOutsideVolume

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.

See Also

IDirectSoundBuffer::SetVolume

IDirectSound3DBuffer::SetMaxDistance

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.

See Also

IDirectSound3DBuffer::GetMaxDistance,
IDirectSound3DBuffer::SetMinDistance

IDirectSound3DBuffer::SetMinDistance

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 m per unit).

See Also

IDirectSound3DBuffer::SetMaxDistance

IDirectSound3DBuffer::SetMode

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.

IDirectSound3DBuffer::SetPosition

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.

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.

IDirectSound3DBuffer::SetVelocity

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.

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 Shift.

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

See Also

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

IDirectSound3DListener

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 parameter manipulation	**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;
```

IDirectSound3DListener::CommitDeferredSettings

The **IDirectSound3DListener::CommitDeferredSetting** 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.

IDirectSound3DListener::GetAllParameters

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.

See Also

IDirectSound3DListener::SetAllParameters

IDirectSound3DListener::GetDistanceFactor

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.

See Also

IDirectSound3DListener::SetDistanceFactor

IDirectSound3DListener::GetDopplerFactor

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.

See Also

IDirectSound3DListener::SetDopplerFactor

IDirectSound3DListener::GetOrientation

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 <= 1.0.

See Also

IDirectSound3DListener::SetOrientation

IDirectSound3DListener::GetPosition

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.

See Also

IDirectSound3DListener::SetPosition

IDirectSound3DListener::GetRolloffFactor

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.

See Also

IDirectSound3DListener::SetRolloffFactor

IDirectSound3DListener::GetVelocity

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).

See Also

IDirectSound3DListener::SetVelocity

IDirectSound3DListener::SetAllParameters

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.

See Also

IDirectSound3DListener::GetAllParameters

IDirectSound3DListener::SetDistanceFactor

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.

See Also

IDirectSound3DListener::GetDistanceFactor

IDirectSound3DListener::SetDopplerFactor

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.

See Also

IDirectSound3DListener::GetDopplerFactor

IDirectSound3DListener::SetOrientation

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::CommitDeferred Settings** 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).

See Also

IDirectSound3DListener::GetOrientation

IDirectSound3DListener::SetPosition

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.

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.

See Also

IDirectSound3DListener::GetPosition

IDirectSound3DListener::SetRolloffFactor

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.

See Also

IDirectSound3DListener::GetRolloffFactor

IDirectSound3DListener::SetVelocity

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.

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).

See Also

IDirectSound3DListener::GetVelocity

IDirectSoundBuffer

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;
```

IDirectSoundBuffer::GetCaps

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.

See Also

DSBCAPS, **DSBUFFERDESC**, **IDirectSoundBuffer**, **IDirectSound::CreateSoundBuffer**

IDirectSoundBuffer::GetCurrentPosition

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetCurrentPosition**

IDirectSoundBuffer::GetFormat

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetFormat**

IDirectSoundBuffer::GetFrequency

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetFrequency**

IDirectSoundBuffer::GetPan

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetPan**, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::GetStatus

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.

See Also

IDirectSoundBuffer

IDirectSoundBuffer::GetVolume

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::Initialize

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.

See Also

DSBUFFERDESC, **IDirectSound::CreateSoundBuffer**, **IDirectSoundBuffer**

IDirectSoundBuffer::Lock

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Unlock**

IDirectSoundBuffer::Play

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.

See Also

IDirectSoundBuffer, **IDirectSound::SetCooperativeLevel**

IDirectSoundBuffer::Restore

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::Lock**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::GetStatus**

IDirectSoundBuffer::SetCurrentPosition

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Play**

IDirectSoundBuffer::SetFormat

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.

If the hardware does not support the requested format, DirectSound may be able to mix the sound in the requested format and then 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.

If the hardware does not support the requested format and DirectSound is unable to emulate it, the call to **SetFormat** fails.

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetFormat**

IDirectSoundBuffer::SetFrequency

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.

See Also

IDirectSoundBuffer, **IDirectSound::CreateSoundBuffer**, **IDirectSoundBuffer::GetFrequency**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::SetFormat**

IDirectSoundBuffer::SetPan

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetPan**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::SetVolume

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetPan**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetPan**

IDirectSoundBuffer::Stop

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::Play**

IDirectSoundBuffer::Unlock

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Lock**

IDirectSoundCapture

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:

CreateCaptureBuffer

GetCaps

Initialize

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;
```

IDirectSoundCapture::CreateCaptureBuffer

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 IDirectSoundCapture::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

IDirectSoundCapture::GetCaps

The **IDirectSoundCapture::GetCaps** method obtains the capabilities of the capture device.

HRESULT IDirectSoundCapture::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

IDirectSoundCapture::Initialize

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 for the DirectSoundCapture object to bind to. 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

IDirectSoundCaptureBuffer

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:

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;
```

IDirectSoundCaptureBuffer::GetCaps

The **IDirectSoundCaptureBuffer::GetCaps** method returns the capabilities of the DirectSound Capture 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

IDirectSoundCaptureBuffer::GetCurrentPosition

The **IDirectSoundCaptureBuffer::GetCurrentPosition** method gets the current capture and reads position 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

IDirectSoundCaptureBuffer::GetFormat

The **IDirectSoundCaptureBuffer::GetFormat** method retrieves the current format of the sound capture buffer.

HRESULT IDirectSoundCaptureBufferGetFormat(
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.

IDirectSoundCaptureBuffer::GetStatus

The **IDirectSoundCaptureBuffer::GetStatus** method retrieves the current status of the sound capture buffer.

HRESULT IDirectSoundCaptureBuffer::GetStatus(
DWORD **lpdwStatus*
);

Parameters

lpdwStatus
Address of a variable to contain the status of the sound 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.

IDirectSoundCaptureBuffer::Initialize

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.

See Also

DSBUFFERDESC, **IDirectSoundCapture::CreateCaptureBuffer**

IDirectSoundCaptureBuffer::Lock

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.

This method accepts an offset and a byte count, and returns two read pointers and their associated sizes. Two pointers are required because sound 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.

HRESULT IDirectSoundCaptureBuffer::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 sound capture buffer is conceptually circular.

lplpvAudioPtr1
Address of a pointer to contain the first block of the sound 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 sound capture buffer to be locked. If the value of this parameter is NULL, the *lplpvAudioPtr1* parameter points to the entire locked portion of the sound 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 NULL 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

IDirectSoundCaptureBuffer::Start

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 IDirectSoundCaptureBuffer::Start(
DWORD *dwFlags*
);

Parameters

dwFlags
Flags that specify the behavior for the capture buffer when capturing sound data. Possible values for **dwFlags** can be one of the following:

DSCBSTART_LOOPING

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

IDirectSoundCaptureBuffer::Stop

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 IDirectSoundCaptureBuffer::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

IDirectSoundCaptureBuffer::Unlock

The **IDirectSoundCaptureBuffer::Unlock** method unlocks the buffer.

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 sound capture buffer does not remain locked for long periods of time.

```
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

IDirectSoundNotify

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;
```

IDirectSoundNotify::SetNotificationPositions

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 IDirectSoundNotify::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.

IKsPropertySet

The **IKsPropertySet** interface is used to get information about extended properties of sound devices, and to manipulate those properties.

The interface is obtained by calling the **QueryInterface** method of an existing interface on a DirectSoundBuffer object. For more information, see DirectSound Property Sets.

The **IKsPropertyS**et method has the following methods:

Get
QuerySupport
Set

Like all COM interfaces, the **IKsPropertySet** interface also inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPKSPROPERTYSET** type is defined as a pointer to the **IKsPropertySet** interface:

```
typedef struct IKsPropertySet    *LPKSPROPERTYSET;
```

IKsPropertySet::Get

The **IKsPropertySet::Get** method retrieves the current value of a property. This method copies the value of the property to the buffer pointed to by *pPropertyValue*.

```
HRESULT Get(
  REFGUID PropertySetId,
  ULONG PropertyId,
  PVOID pPropertyParams,
  ULONG cbPropertyParams,
  PVOID pPropertyData,
  ULONG cbPropertyData,
  PULONG pcbReturnedData

);
```

Parameters

PropertySetId
The property set.

PropertyId
The property within the property set.

pPropertyParams
Pointer to instance parameters for the property.

cbPropertyParams
The number of bytes in the structure pointed to by *pPropertyParams*.

pPropertyData
Pointer to buffer in which to store the value of the property.

cbPropertyData
The number of bytes in the structure pointed to by *pPropertyData*.

pcbReturnedData
The number of bytes returned in the structure pointed to by *pPropertyData*.

Return Values

Return values are determined by the designer of the property set.

IKsPropertySet::QuerySupport

The **IKsPropertySet::QuerySupport** method determines whether a property set is supported by the object.

```
HRESULT QuerySetSupport(
  REFGUID PropertySetId,
  ULONG PropertyId,
  PULONG pSupport
);
```

Parameters

PropertySetId
Identifier of the property set.

PropertyId
Identifier of the property within the property set.

pSupport
Pointer to a **ULONG** in which to store flags indicating the support provided by the driver. The following values may be set:

KSPROPERTY_SUPPORT_GET
The driver sets this flag to indicate that the property can be read by using the **IKsPropertySet::Get** method.

KSPROPERTY_SUPPORT_SET
The driver sets this flag to indicate that the property can be changed by using the **IKsPropertySet::Set** method.

Return Values

Returns E_NOTIMPL for property sets that are not supported. Other return values are determined by the designer of the property set.

Remarks

Whether it is valid to support some properties within the set but not others depends on the definition of the property set. Consult the hardware manufacturer's specification for the property set of interest.

IKsPropertySet::Set

The **IKsPropertySet::SetProperty** method sets the current value of a property to the value stored in the buffer pointed to by *pPropertyValue*.

```
HRESULT SetProperty(
  REFGUID PropertySetId,
  ULONG PropertyId,
  PVOID pPropertyParams,
  ULONG cbPropertyParams,
  PVOID pPropertyData,
  ULONG cbPropertyData
);
```

Parameters

PropertySetId
The property set.

PropertyId
The property within the property set.

pPropertyParams
Pointer to instance parameters for the property.

cbPropertyParams
Size of the structure pointed to by *pPropertyParams*.

pPropertyData
Address of a buffer containing the value to which to set the property.

cbPropertyData
Size of the structure pointed to by *pPropertyData*.

Return Values

Return values are determined by the designer of the property set.

Functions

This section contains reference information for the following DirectSound and DirectSoundCapture global functions:

- **DirectSoundCreate**
- **DirectSoundEnumerate**
- **DirectSoundCaptureCreate**
- **DirectSoundCaptureEnumerate**

DirectSoundCreate

The **DirectSoundCreate** function creates and initializes an **IDirectSound** interface.

```
HRESULT DirectSoundCreate(
  LPGUID lpGuid,
  LPDIRECTSOUND * ppDS,
  IUnknown FAR * 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.

See Also

IDirectSound::GetCaps, **IDirectSound::SetCooperativeLevel**

DirectSoundEnumerate

The **DirectSoundEnumerate** function enumerates the DirectSound drivers installed in the system.

```
BOOL 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.

See Also

DSEnumCallback

DirectSoundCaptureCreate

The **DirectSoundCaptureCreate** function creates and initializes an object that supports the **IDirectSoundCapture** interface.

HRESULT 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

DirectSoundCaptureEnumerate

The **DirectSoundCaptureEnumerate** function enumerates the DirectSoundCapture objects installed in the system.

BOOL 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.

Callback Function

This section contains reference information for the following DirectSound callback function.

- **DSEnumCallback**

DSEnumCallback

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 DSEnumCallback(
LPGUID *lpGuid***,**
LPCSTR *lpstrDescription***,**
LPCSTR *lpstrModule***,**
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 *lpstrDescription* and *lpstrModule* 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.

See Also

DirectSoundEnumerate

Structures

This section contains reference information for the following structures used with DirectSound:

- **DS3DBUFFER**
- **DS3DLISTENER**
- **DSBCAPS**
- **DSBPOSITIONNOTIFY**
- **DSBUFFERDESC**
- **DSCAPS**
- **DSCBCAPS**
- **DSCBUFFERDESC**
- **DSCCAPS**

DS3DBUFFER

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** structurethat 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.

DS3DLISTENER

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.

DSBCAPS

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, that 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.

See Also

IDirectSound::CreateSoundBuffer, **IDirectSoundBuffer::GetCaps**

DSBPOSITIONNOTIFY

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.

DSBUFFERDESC

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**.

See Also

IDirectSound::CreateSoundBuffer

DSCAPS

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 (those located in on-board sound memory). 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.

See Also

IDirectSound::GetCaps

DSCBCAPS

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 NULL 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.

DSCBUFFERDESC

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 NULL 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.

DSCCAPS

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
No flags are currently defined.

dwFormats
Standard formats that are supported. See the reference for the **WAVEINCAPS** structure in the Platform SDK.

dwChannels

Number specifying the number of channels supported by the device, where 1 is mono, 2 is stereo, and so on.

Return Values

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.