Chapter 3

Microsoft® DirectX™ 3 Software Development Kit

DirectSound

DirectSound

CHAPTER 3

About DirectSound

The Microsoft® DirectSound® application programming interface (API) is the audio component of the DirectX™ 3 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 Windows®-based applications and device drivers.

DirectX 3 allows you access to the display and audio hardware while insulating you from the specific details of that hardware. The overriding design goal in DirectX 3 is speed. Instead of providing a high-level set of functions, DirectSound provides a device-independent interface, allowing applications to take full advantage of the capabilities of the audio hardware.

DirectSound Architecture

This section contains general information about the relationship between the DirectSound component and the rest of DirectX, the operating system, and the system hardware. The following topics are discussed:

- *Architectural Overview*
- *Object Types*
- *Software Emulation*
- *Device Drivers*
- *Cooperative Levels*
- *System Integration*

Architectural Overview

Programming for high-performance applications and games requires efficient and dynamic sound production. Microsoft provides two methods for achieving this: MIDI streams and DirectSound. MIDI streams are actually part of the Windows 95 multimedia API. They provide the ability to time stamp MIDI messages and send a buffer of these messages to the system, which can then efficiently integrate them with its processes. For more information about MIDI streams, see the documentation included with the Microsoft Win32® Software Development Kit (SDK).

DirectSound implements a new model for playing digitally recorded sound samples and mixing sample sources. As with other object classes in the DirectX 3 SDK, DirectSound uses the hardware to its greatest advantage whenever possible, and it emulates hardware features in software when the feature is not present in

hardware. You can query hardware capabilities at run time to determine the best solution for any given personal computer configuration.

DirectSound is built on the *IDirectSound* and *IDirectSoundBuffer* COM-based interfaces, and it is extensible to other interfaces. For more information about COM concepts that you should understand to create applications with the DirectX 3 SDK, see *The Component Object Model*.

The DirectSound object represents the sound card and its various attributes. An application creates the DirectSoundBuffer object by using the DirectSound object's **IDirectSound::CreateSoundBuffer** method. The DirectSoundBuffer object represents a buffer containing sound data. Several DirectSoundBuffer objects can exist and be mixed together in the primary DirectSoundBuffer object. DirectSound buffers are used to start, stop, and pause sound playback, as well as to set attributes such as frequency, format, and so on.

Depending on the card type, DirectSound buffers can exist in hardware as onboard 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.

The primary sound buffer is typically used to mix sound from secondary sound buffers, but it can be accessed directly for custom mixing or other specialized activities. (Use caution in locking the primary buffer, because it blocks all access to the sound hardware from other sources.)

The secondary buffers can store common sounds played throughout an application, such as in a game. The application can play a sound stored in a secondary buffer as a single event or as a looping sound that plays repeatedly.

Secondary buffers can also play sounds that are larger than the available sound-buffer memory. When the secondary buffer is used to play a sound that is larger than the sound-buffer memory, the secondary buffer serves as a queue that stores the portions of the sound about to be played.

Object Types

The most fundamental type of object is the DirectSound object, which represents the sound card itself. The *IDirectSound* Component Object Model (COM) interface controls the DirectSound object; the methods of this interface allow the application to change the characteristics of the card.

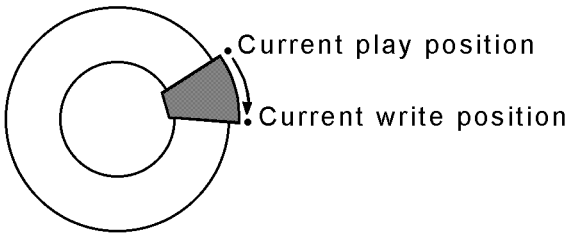
The second type of object is a sound buffer. DirectSound uses *primary* and *secondary sound buffers*. Primary sound buffers represent the audio data that is actually heard by the user, while secondary sound buffers represent individual source sounds. DirectSound provides controls for primary and secondary sound buffers in the *IDirectSoundBuffer* interface.

Primary buffers control sound characteristics, such as output format and total volume. Also, your application can write directly to the primary buffer. In this case, however, the DirectSound mixing and hardware-acceleration features are not available. In addition, writing directly to the primary buffer can interfere with other DirectSound applications. When possible, your application should write to secondary buffers instead of the primary buffer. Secondary buffers allow the system to emulate features that might not be present in the hardware; they also allow an application to share the sound card with other applications in the system.

Secondary buffers represent single sound sources that an application uses. An application can play or stop each buffer independently. DirectSound mixes all playing buffers into the primary buffer, and then it outputs the primary buffer to the sound device. Secondary buffers can reside in hardware or system buffers; hardware buffers are mixed by the sound device without any system-processing overhead.

Secondary sound buffers can be either *static* or *streaming sound buffers*. A static sound buffer means that the buffer contains an entire sound. A streaming sound buffer means that the buffer contains only part of a sound, and, therefore, your application must continually write new data to the buffer while it is playing. DirectSound attempts to store static buffers by using sound memory located on the sound hardware, if available. Buffers stored on the sound hardware do not consume system processing time when they are played because the mixing is done in the hardware. Reusable sounds, such as gunshots, are perfect candidates for static buffers.

Your applications will work with two significant positions within a sound buffer: the current play position and the current write position. The current play position indicates the location in the buffer where the sound is being played. The current write position indicates the location where you can safely change the data in the buffer. The following illustration shows the relationship between these two positions.



Although DirectSound buffers are conceptually circular, they are implemented by using contiguous, linear memory. When the current play position reaches the end of the buffer, it wraps back to the beginning of the buffer.

Note

This section discusses the DirectSound object, the DirectSoundBuffer object, and how your applications can use these objects.

- *The DirectSound Object*
- *The DirectSoundBuffer Object*

The DirectSound Object

Each sound device installed in the system is represented by a DirectSound object that is accessed through the *IDirectSound* interface. Your application can create a DirectSound object by calling the **DirectSoundCreate** function that returns an **IDirectSound** interface. DirectSound objects installed in the system can be enumerated by calling the **DirectSoundEnumerate** function.

Windows is a multitasking operating system. Typically, users run several programs at once and expect them all to share resources. DirectSound objects share sound devices by tracking the input focus. They produce sound only when their owning application has the input focus. When an application loses the input focus, the audio streams from that object are muted. 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.

The header file for DirectSound includes C programming macro definitions for the methods of the **IDirectSound** and *IDirectSoundBuffer* interfaces.

The DirectSoundBuffer Object

Each sound or audio stream is represented by a DirectSoundBuffer object that your application can access through the *IDirectSoundBuffer* interface. An application can create a DirectSoundBuffer object by calling the **IDirectSound::CreateSoundBuffer** method, which returns an **IDirectSoundBuffer** interface.

Applications can also create primary sound buffers or secondary sound buffers. In the current implementation, each DirectSound object has only one primary buffer.

Your application can write data into sound buffers by locking the buffer, writing data to the buffer, and then unlocking the buffer. A buffer can be locked by calling the **IDirectSoundBuffer::Lock** method. This method returns a pointer to the locked portion of the buffer. After the buffer is locked, your application can copy audio data to the buffer. After writing data to the buffer, you must unlock the buffer and complete the write operation by calling the **IDirectSoundBuffer::Unlock** method.

The primary sound buffer contains the data that is heard. You can play audio data from a secondary sound buffer by using the **IDirectSoundBuffer::Play** method. This method causes DirectSound to begin mixing the secondary buffer into the primary buffer. By default, **IDirectSoundBuffer::Play** plays the buffer once and stops at the end. You can also play a sound repeatedly in a continuous loop by specifying the DSBPLAY_LOOPING flag when calling this method. The application can stop a buffer that is playing by using the **IDirectSoundBuffer::Stop** method.

Typically, the duration of a sound determines how your application uses the associated sound buffer. If the sound data is only a few seconds long, you can use a static sound buffer to store the sound. If the sound is longer than that, you should use a streaming sound buffer.

Your application can create a DirectSoundBuffer object that has a static buffer by using the **IDirectSound::CreateSoundBuffer** method and specifying the DSBCAPS_STATIC flag. (If your application does not specify this flag, a streaming sound buffer is created.) DirectSound attempts to store static buffers by using sound memory located on the sound hardware, if that memory is available. Buffers stored on sound hardware do not consume system processing time when they are played because the mixing is done in the hardware. Reusable sounds, such as engine roars, cheers, and jeers, are perfect candidates for static buffers.

Streaming buffers can also use hardware mixing if it is supported by the sound device; however, this is efficient only when your application runs on computers with fast data buses, such as the peripheral component interconnect (PCI) bus. If the computer does not have a fast bus, the data-transfer overhead outweighs the benefits of hardware mixing. DirectSound locates streaming buffers in hardware only if the sound device is located on a fast bus.

Software Emulation

DirectSound can emulate in software the features that a particular sound card does not directly support without loss of functionality. Applications can query DirectSound to determine the capabilities of the audio hardware by using the **IDirectSound::GetCaps** method. A high-performance game, for example, can use this information to scale its audio features.

Device Drivers

DirectSound accesses the sound hardware through the DirectSound hardware-abstraction layer (HAL), an interface that is implemented by the audio-device driver. This is a Windows audio-device driver that has been modified to support the HAL. This driver architecture provides backward compatibility with existing Windows-based applications. 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
- Fails the operation request 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 fails the request and DirectSound emulates the operation.

If a DirectSound driver is not available, DirectSound communicates with the audio hardware through the standard Windows 95, Windows NT®, or Windows 3.1 audio-device driver. In this case, all DirectSound features are still available through software emulation, but hardware acceleration is not possible.

Cooperative Levels

DirectSound defines four cooperative levels for sound devices: normal, priority, exclusive, and write-primary. Applications set a sound device's cooperative level by using the **IDirectSound::SetCooperativeLevel** method. Applications can create global or sticky sound buffers in all cooperative levels except write-primary.

The normal cooperative level is the lowest level. At the normal level, the **IDirectSoundBuffer::SetFormat** and **IDirectSound::Compact** methods cannot be called. In addition, the application cannot obtain write access to primary buffers. All applications using this cooperative level use a primary buffer format of 22 kHz, stereo sound, and 8-bit samples to make task switching as smooth as possible.

When using a DirectSound object with the priority cooperative level, the application has first priority to hardware resources, such as hardware mixing, and can call **IDirectSoundBuffer::SetFormat** and **IDirectSound::Compact**.

When using a DirectSound object with the exclusive cooperative level, the application has all the privileges of the priority level. In addition, however, when the application has the input focus, its buffers are the only ones that are audible. After the input focus is gained, DirectSound restores the application's preferred wave format, which was defined in the most recent call to **IDirectSoundBuffer::SetFormat**. (DirectSound restores the wave format regardless of the priority level.)

The highest cooperative level is write-primary. When using a DirectSound object with this cooperative level, your application has direct access to the primary sound buffer. In this mode, the application must lock the buffer by using the

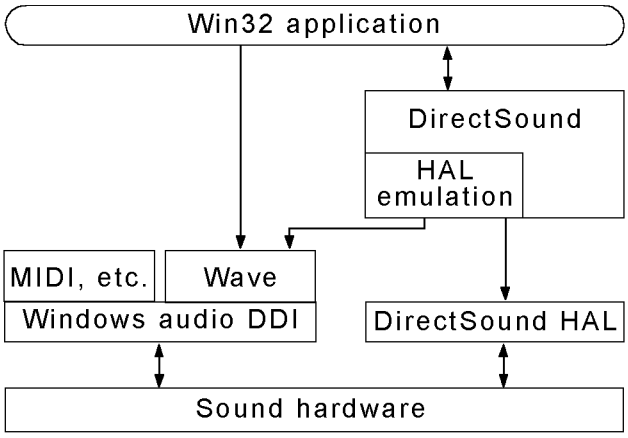
IDirectSoundBuffer::Lock method and write directly to the primary buffer. While this occurs, secondary buffers cannot be played.

When the application is set to the write-primary cooperative level and gains the input focus, all secondary buffers for other applications are stopped and marked as lost. (These buffers must be restored by using the **IDirectSoundBuffer::Restore** method before they can be played again.) When this application, in turn, loses the input focus, its primary buffer is marked as lost. It also can be restored after the application regains the input focus.

The write-primary level is not required to create a primary buffer. However, to obtain access to the audio samples in the primary buffer, the application must be set to the write-primary level. If the application is not set to this level, then all calls to **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Play** fail, although some methods, such as **IDirectSoundBuffer::GetFormat**, **IDirectSoundBuffer::SetFormat**, and **IDirectSoundBuffer::GetVolume**, can still be called successfully.

System Integration

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



Using a device driver for the sound hardware that implements the DirectSound HAL provides the best performance for playing audio. The device driver implements each function of the HAL to leverage the architecture of the sound hardware and provide functionality and high performance. The HAL describes the capabilities of the hardware to DirectSound and passes requests from DirectSound to the hardware. If the hardware cannot handle the request, the driver causes the call to fail. DirectSound then emulates the request in software.

Note

Your application can use DirectSound features even when no DirectSound driver is present. If the sound hardware does not have an installed DirectSound driver, DirectSound uses its HAL emulation layer. This layer uses the Windows multimedia waveform-audio functions.

The DirectSound functions and the 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 your application must use both sets of functions, you should use each set sequentially. For example, you could open the sound hardware by using the **DirectSoundCreate** function, play sounds by using the *IDirectSound* and *IDirectSoundBuffer* interfaces, and close the sound hardware by using the **IDirectSound::Release** method. The sound hardware would then be available for the waveform-audio functions of the Win32 SDK.

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

The waveform-audio functions continue to be a practical solution for certain applications. For example, your application can easily play a single sound or audio stream, such as an introductory sound, by using the **PlaySound** or **WaveOut** function.

Microsoft Video for Windows currently uses the waveform-audio functions to output 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 Overview

This section contains general information about the DirectSound component. The following topics are discussed:

- *DirectSound Features*
- *Three-Dimensional Sound*

DirectSound Features

This section describes DirectSound's low-latency audio mixing and its ability to take advantage of accelerated sound hardware. In addition, this section discusses what you should consider when you design applications that write to a primary sound buffer.

- *Mixing*
- *Hardware Acceleration*
- *Write Access to the Primary Buffer*

Mixing

The most frequently used feature of DirectSound is the mixing of audio streams with little *latency*. Latency is the delay between the time that a sound buffer plays and the time that the speakers reproduce the sound. Your application can create one or more secondary sound buffers and write audio data to them. You can choose to play or stop any of these buffers. DirectSound mixes all playing buffers and writes the result to the primary sound buffer, which supplies the sound hardware with audio data. Only the available processing time limits the number of buffers that DirectSound can mix.

A user perceives no delay between the time that a buffer plays and the time that the speakers reproduce the sound when the latency is 20 milliseconds or less. The DirectSound mixer provides 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 synchronously. However, if DirectSound must use the HAL emulation layer (if a DirectSound driver for the sound hardware is not present), the mixer cannot achieve low latency and a hardware-dependent delay (typically 100-150 milliseconds) occurs before the sound is reproduced.

Only buffers from a single application are audible at any given instance because only one application at a time can open a particular DirectSound device,

Hardware Acceleration

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 to receive a full description of the hardware capabilities of the sound device. From this information, you can specify which sound buffers should receive hardware acceleration.

Because your application determines when to use each effect, when to play each sound buffer, and what priority each buffer should take, it can allocate hardware resources as it needs them.

Write Access to the Primary Buffer

The primary sound buffer outputs audio samples to the sound device. DirectSound provides direct write access to the primary buffer; however, this feature is useful for a very limited set of applications that require specialized mixing or other effects not supported by secondary buffers. Gaps in sound are difficult to avoid when an application writes directly to the primary buffer. Applications that access the primary buffer directly are subject to stringent performance requirements.

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 previous block in the buffer from repeating. During buffer creation, you cannot specify the size of the buffer, and you must accept the returned size after the buffer is created.

When you obtain write access to a primary sound buffer, other DirectSound features become unavailable. Secondary buffers are not mixed and, consequently, hardware-acceleration mixing is unavailable. (When DirectSound mixes sounds from secondary buffers, it places the mixed audio data in the primary buffer.)

Most of your 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 your 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.

Three-Dimensional Sound

DirectSound enables an application to change the apparent position of a sound source, by using the *IDirectSound3DBuffer* and *IDirectSound3DListener* interfaces. Sound sources 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.

The following topics are discussed in this overview of 3D sound:

- *Perception of Sound Positions*
- *Listeners*

- *Sound Cones*
- *Minimum and Maximum Distances*
- *Position Versus Velocity*
- *Integration with Direct3D*
- *Units of Measure and Distance Factors*
- *Mono and Stereo Sources*

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 listener's right ear slightly before it arrives at the left ear. (The duration of this offset is approximately a millisecond.)
- *Muffling.* The orientation of people's ears ensures that sounds coming from behind the listener are slightly muffled compared with sounds coming from in front of the listener. In addition, if a sound is coming from the listener's right, the sounds reaching the left ear will be muffled by the mass of the listener's head.

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 DirectSound's positioning system. When hardware that supports 3D 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 listener's shoulders and external ear parts.

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, for example, the sound will appear to have gone by the listener without much help from subtle cues.

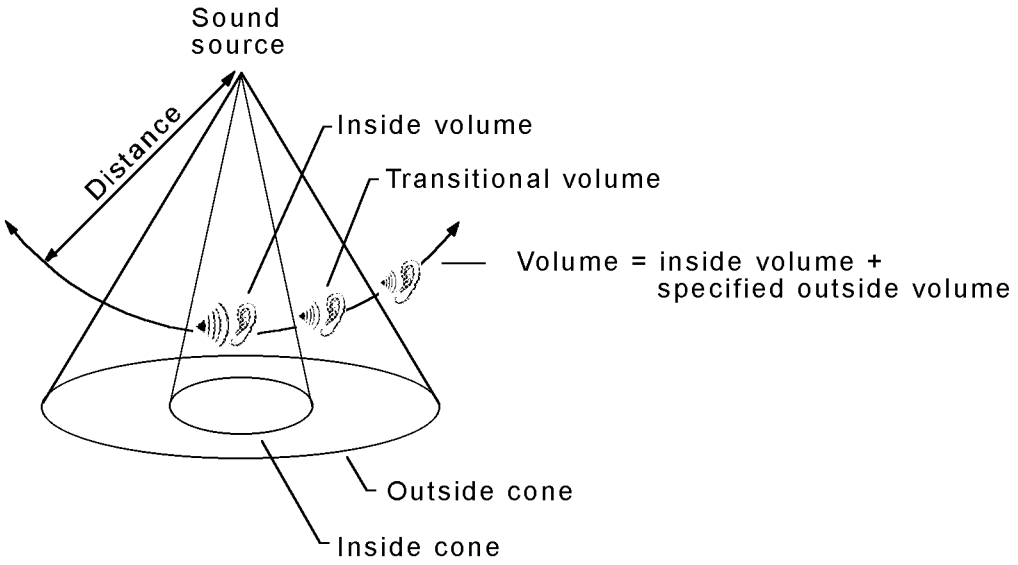
Listeners

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 to positions of every other object in a scene. DirectSound exposes this capability through the *IDirectSound3DListener* interface.

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 -10,000, 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:

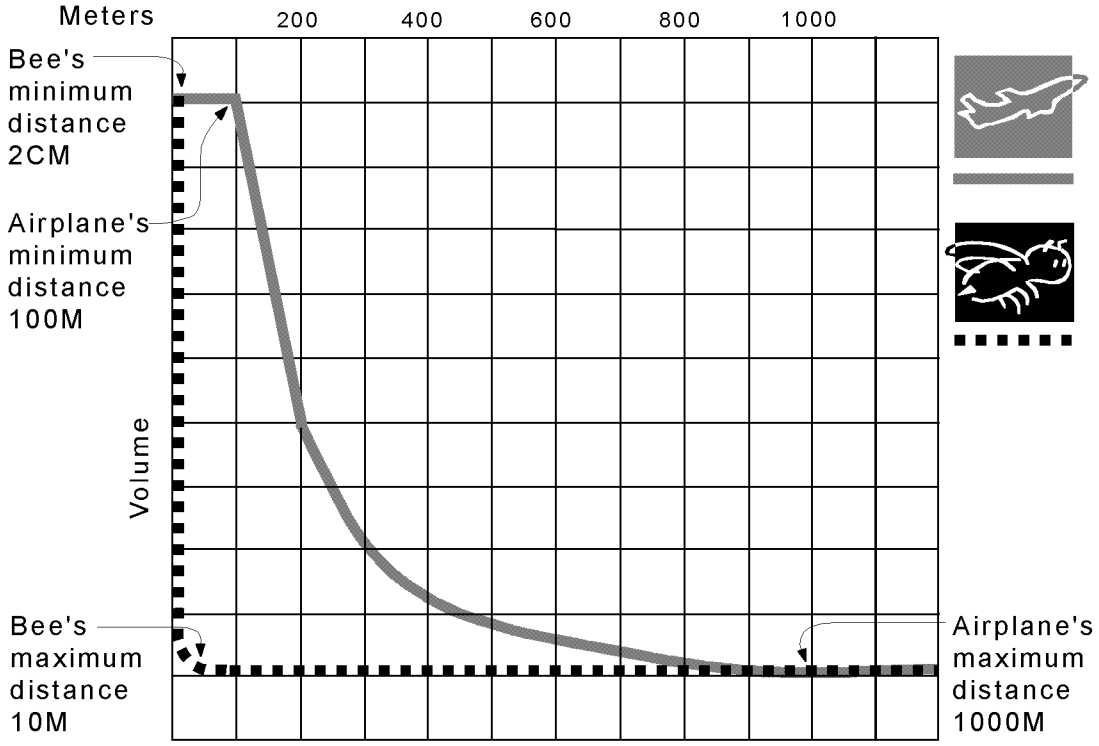


Technically, every sound buffer represented by the *IDirectSound3DBuffer* interface is a sound cone, but often these sound cones behave like omnidirectional sound sources. For example, the default value for the volume outside the sound cone is zero; unless the application changes this value, the volume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

Minimum and Maximum Distances

As a listener approaches a sound source, the sound gets louder. Past a certain point, however, it isn't 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:



Position Versus Velocity

Every 3D sound buffer and 3D listener has both a position and a velocity. From a graphics and animation standpoint, these characteristics seem quite similar. As one would expect, the position of a 3D sound buffer or listener represents its location in 3D space. However, the velocity of that buffer or listener does not represent how fast the object is moving in space. Rather, DirectSound uses the velocity of a given buffer or listener to calculate Doppler-shift effects.

Velocity adjustments can be useful if you would like to exaggerate the Doppler shift of an object. For example, imagine that you want the sound of an oncoming race car to seem as though it is screaming past the listener. If you exaggerate the effects of Doppler shift for the listener, the exaggeration will affect all sound buffers that the listener hears. To exaggerate the effect for the race car alone, you could increase the velocity setting for the car's 3D sound buffer.

The system handles cumulative Doppler-shift effects for you. If your application's listener and the sound source have velocities, the system automatically calculates the relationship between the velocities and adjusts the Doppler effect accordingly.

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 also employed by DirectSound. (For information about coordinate systems, see *3D Coordinate Systems*, in the Direct3D overview material.)

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

Units of Measure and Distance Factors

The default values for the 3D 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.

DirectSound's 3D effects 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 that represents 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.

Mono and Stereo Sources

Stereo sound sources are not particularly useful in 3D sound environments. In effect, a stereo signal consists of two separate monaural tracks played simultaneously on different speakers.

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

DirectSound Interface Overviews

This section contains general information about the following DirectSound interfaces:

- *IDirectSound Interface*
- *IDirectSound3DBuffer Interface*
- *IDirectSound3DListener Interface*
- *IDirectSoundBuffer Interface*

IDirectSound Interface

A DirectSound object describes the audio hardware on a system. The audio data itself resides in a buffer called a DirectSoundBuffer object. For more information about DirectSound buffers, see *IDirectSoundBuffer Interface*. The *IDirectSound* interface enables your application to define and control the sound card, speaker, and memory environment.

This section describes how your application can retrieve the capabilities of the system's sound device, create sound buffers, set the configuration of the system's speakers, and compact hardware memory.

- *Device Capabilities*
- *Creating Buffers*
- *Speaker Configuration*
- *Hardware Memory Management*

Device Capabilities

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. For optimal performance, you should make this call to determine the capabilities of the resident sound card, then modify its sound parameters as appropriate.

Creating Buffers

After calling the **DirectSoundCreate** function to create a DirectSound object and investigating the capabilities of the sound device, your application can create and enumerate the sound buffers that contain audio data. The **IDirectSound::CreateSoundBuffer** method creates a sound buffer. The **IDirectSound::DuplicateSoundBuffer** method creates a second sound buffer using the same physical buffer memory as the first. If you duplicate a sound buffer, you can play both buffers independently without wasting buffer memory.

Your application must use the **IDirectSound::SetCooperativeLevel** method to set its cooperative level for a sound device before playing any sound buffers. Most applications use a standard priority level, DSSCL_NORMAL, which ensures they will not conflict with other applications.

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**. Currently recognized configurations include headphones, binaural headphones, stereo, quadraphonic, and surround sound.

Hardware Memory Management

Your application can use the **IDirectSound::Compact** method to move any onboard sound memory into a contiguous block to make the largest portion of free memory available.

IDirectSound3DBuffer Interface

The *IDirectSound3DBuffer* interface provides access to the 3D parameters of a sound buffer. This interface is not supported by all sound buffers.

This section describes how your applications can obtain a pointer to an **IDirectSound3DBuffer** interface and manage buffer parameters by using interface methods. The following topics are discussed:

- *Obtaining an IDirectSound3DBuffer Interface Pointer*
- *Batch Parameter Manipulation*

Note

- *Minimum and Maximum Distance Values*
- *Operation Mode*
- *Position and Velocity*
- *Sound Projection Cones*

Obtaining an IDirectSound3DBuffer Interface Pointer

To obtain a pointer to an *IDirectSound3DBuffer* interface, you must first create a secondary 3D sound buffer. Do this by using the **IDirectSound::CreateSoundBuffer** method, specifying the DSBCAPS_CTRL3D flag 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.

```
// lpDsbSecondary was created with DSBCAPS_CTRL3D.
hr = lpDsbSecondary->QueryInterface(IID_IDirectSound3DBuffer,
        &lpDs3dBuffer);
    if(SUCCEEDED(hr)) {
    // Set 3D parameters of this sound.
       .
       .
       .
}
```

DirectSound supports monaural or a stereo wave format for 3D sound buffers. However, 3D positional sound works best with monaural sounds because the 3D processing creates a 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 3D processing is applied. To position the two stereo channels at different locations, the application must divide the stereo stream into two monaural streams, and write this data into two monaural sound buffers.

Pan control conflicts with 3D processing. Therefore, if both DSBCAPS_CTRL3D and DSBCAPS_CTRLPAN are specified, DirectSound causes a creation request to fail.

Batch Parameter Manipulation

Applications can retrieve or set a 3D 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 that describe the buffer at once. An application can perform a batch parameter manipulation in a single call by using the

IDirectSound3DBuffer::GetAllParameters and **IDirectSound3DBuffer::SetAllParameters** methods.

Minimum and Maximum Distance Values

Applications can specify minimum and maximum distance values for a 3D sound buffer. The minimum distance is the distance at which the sound does not get louder. Conversely, the maximum distance is the distance at which the sound no longer attenuates. For more information about the relationship between these values, see *Minimum and Maximum Distances*.

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.

Operation Mode

Sound buffers have three processing modes: normal, head-relative, and disabled. Normal processing mode 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, so that the relative parameters remain constant. In the disabled mode, 3D 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 3D 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*.

Position and Velocity

An application can set and retrieve a 3D sound buffer's position in 3D 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, use the **IDirectSound3DBuffer::SetVelocity** and **IDirectSound3DBuffer::GetVelocity** methods. A buffer's position is not affected by its velocity. For more information about the relationship between position and velocity, see *Position Versus Velocity*.

Sound Projection Cones

A 3D sound buffer has two sound cones: an inside cone and an outside cone. An application can set and retrieve the cone angles, maximum and minimum

distances, and position and orientation of a buffer's sound projection cones by using various *IDirectSound3DBuffer* methods. For more information about the behavior and characteristics of sound projection cones, see *Sound Cones*.

Designing sound cones properly can add dramatic effects to your application. For example, imagine that you want to use the sound of a ghostly voice. Instead of simply playing the sound, you could create additional suspense in your application by using **IDirectSound3DBuffer** methods. Position the sound source in the center of a room, setting its orientation toward a door. Then, focus the sound cones to the width of the door and set the outside cone volume to -10,000 (inaudible). These characteristics, when combined, make it seem that the voice is emanating from a room that the user passes by.

This section describes how to set the following sound characteristics:

- Cone Angles and Cone Orientation
- Inside and Outside Cone Volumes

Cone Angles and Cone Orientation

An application sets or retrieves the angles that define 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 cone. The outside cone angle must always be equal to or greater than the inside cone angle.

Inside and Outside Cone Volumes

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 until this parameter is changed.

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. When the listener is outside the sound cone, the cone outside volume is applied as well, making the

actual volume the sum of the two. Near the boundary of the cone, the volume fades smoothly between the two levels to avoid perceptual artifacts.

IDirectSound3DListener Interface

A 3D listener represents the person who hears sounds generated by sound buffer objects in 3D space. The *IDirectSound3DListener* interface controls the listener's position and apparent velocity in 3D 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 an IDirectSound3DListener Interface Pointer*
- *Batch Parameter Manipulation*
- *Deferred Settings*
- *Distance Factor*
- *Doppler Factor*
- *Listener Position and Velocity*
- *Listener Orientation*
- *Rolloff Factor*

Obtaining an IDirectSound3DListener Interface Pointer

To obtain a pointer to an *IDirectSound3DListener* interface, you must first create a primary 3D sound buffer. Do this by using the **IDirectSound::CreateSoundBuffer** method, specifying the DSBCAPS_CTRL3D flag 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:

```
// lpDsbPrimary was created by using DSBCAPS_CTRL3D.

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

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

```
    .
    .
}
```

Batch Parameter Manipulation

Applications can retrieve or set a 3D 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

Every change to a 3D listener parameter requires a recalculation of the 3D positional filter parameters. Therefore, to get maximum efficiency, your application can make parameter changes while using the DS3D_DEFERRED flag in the *dwApply* parameter of the applicable method. It can then call **IDirectSound3DListener::CommitDeferredSettings** when all settings are complete.

Any deferred settings are overwritten if your application calls the same setting with the DS3D_IMMEDIATE flag before it calls **IDirectSound3DListener::CommitDeferredSettings**. That is, if you set the listener velocity to (1,2,3) by using 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 information about distance factors, see *Units of Measure and Distance Factors*.

To set a distance factor for an application that uses feet, for example, call the **IDirectSound3DListener::SetDistanceFactor**, specifying .30480006096 as the *flDistanceFactor* parameter. (This value is the number of meters in a foot.) After an application sets the distance factor, it can call any methods that apply to that listener, using the application's native distance unit.

Consequently, an application can retrieve the current distance factor set for a listener by using the **IDirectSound3DListener::GetDistanceFactor** method. The default value is DS3D_DEFAULTDISTANCEFACTOR (1.0), meaning that one 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 right, 7.2 m above, and 20.9 m behind the origin. If the distance factor were changed to 2.0, the same position vector would mean that the object is 6.0 m right, 14.4 m above, and 41.8 m behind the origin.

Doppler Factor

DirectSound applies Doppler-shift effects to sounds, based on the listener's velocity in relation to one or more 3D sound buffers. DirectSound can apply to a sound up to 10 times the Doppler shift experienced in the real world by setting a Doppler factor. To set this factor, use the **IDirectSound3DListener::SetDopplerFactor** method. The Doppler factor can range from 0 to 10. A value of 0 means no Doppler shift is applied to a sound. Every other value represents a multiple of the Doppler shift experienced in the real world. In other words, a value of 1 means the Doppler shift experienced in the real world is applied to the sound; a value of 2 means two times the Doppler shift experienced in the real world, and so on. To retrieve the Doppler factor set for a 3D listener, use the **IDirectSound3DListener::GetDopplerFactor** method.

Listener Position and Velocity

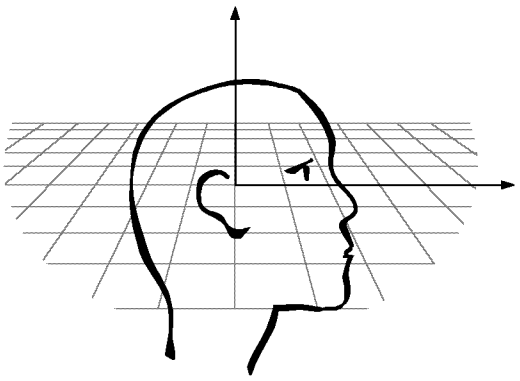
An application can set and retrieve a listener's position in 3D space by using the **IDirectSound3DListener::SetPosition** and **IDirectSound3DListener::GetPosition** methods.

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. For more information about the relationship between position and velocity, see *Position Versus Velocity*.

Listener Orientation

The listener's orientation plays a strong role in 3D 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*.

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:



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).

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 apply to a sound up to 10 times the rolloff experienced in the real world by setting a rolloff factor. To set this factor, use the **IDirectSound3DListener::SetRolloffFactor** method. The rolloff factor can range from 0 to 10. A value of 0 means no rolloff is applied to a sound. Every other value represents a multiple of the rolloff experienced in the real world. In other words, a value of 1 means the rolloff experienced in the real world is applied to the sound; a value of 2 means two times the rolloff experienced in the real world, and so on. To retrieve the rolloff factor, use the **IDirectSound3DListener::GetRolloffFactor** method.

IDirectSoundBuffer Interface

The *IDirectSoundBuffer* interface enables your application to work with buffers of audio data. Audio data resides in a DirectSound buffer. Your application creates DirectSound buffers for each sound or audio stream to be played.

The primary sound buffer represents the actual audio samples sent to the sound device. These samples can be a single audio stream or the mixed output of several audio streams. Applications typically do not directly access the audio data in a primary sound buffer. However, the primary buffer can be used for control purposes, such as setting the output volume or wave format.

A secondary sound buffer represents a single output stream or sound. Your application can play this buffer into the primary sound buffer. Secondary sound

Note

buffers that play concurrently are mixed into the primary buffer, which is then sent to the sound device.

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.

This section discusses how your application can manage sound-buffer playback; track and control volume, frequency, and pan settings; retrieve sound-buffer information; and manage memory.

- *Play Management*
- *Sound-Environment Management*
- *Retrieving Information*
- *Memory Management*

Play Management

Your application can use the **IDirectSoundBuffer::Play** and **IDirectSoundBuffer::Stop** methods to control the real-time playback of sound. The application can also play a sound by using **IDirectSoundBuffer::Play**. The buffer stops automatically when its end is reached. However, if the application specifies looping, the buffer repeats until the application calls **IDirectSoundBuffer::Stop**.

The **IDirectSoundBuffer::Lock** method retrieves a write pointer to the current sound buffer. After writing audio data to the buffer, you must unlock the buffer by using the **IDirectSoundBuffer::Unlock** method. You should not leave the buffer locked for extended periods.

To retrieve or set the current position in the sound buffer, call the **IDirectSoundBuffer::GetCurrentPosition** or **IDirectSoundBuffer::SetCurrentPosition** method.

Sound-Environment Management

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.

Note

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.

Retrieving Information

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 set the format of the sound data in the primary sound buffer.

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

Memory Management

Your application can use the **IDirectSoundBuffer::Restore** method to restore the sound-buffer memory for a specified DirectSoundBuffer object. Although this is useful if the buffer has been lost, the **IDirectSoundBuffer::Restore** method restores only the memory itself. It cannot restore the contents of the memory. After the buffer memory is restored, the application must write valid sound data to it.

DirectSound Examples

Your application should follow these basic steps to implement DirectSound:

1. Create a DirectSound object by calling the **DirectSoundCreate** function.
2. Specify a cooperative level by calling the **IDirectSound::SetCooperativeLevel** method. Most applications use the lowest level, DSSCL_NORMAL.
3. Create secondary sound buffers by using the **IDirectSound::CreateSoundBuffer** method. Your application need not specify in the **DSBUFFERDESC** structure that these buffers are secondary buffers; the creation of secondary buffers is the default.
4. Load the secondary buffers with data. Use the **IDirectSoundBuffer::Lock** method to obtain a pointer to the data area and the **IDirectSoundBuffer::Unlock** method to set the data to the device.
5. Use the **IDirectSoundBuffer::Play** method to play the secondary buffers.

6 Stop all buffers when your application has finished playing sounds by using the **IDirectSoundBuffer::Stop** method of the DirectSoundBuffer object.

7 Release the secondary buffers.

8 Release the DirectSound object.

Your application can also perform the following optional items:

- Set the output format of the primary sound buffer by creating a primary buffer and calling the **IDirectSoundBuffer::SetFormat** method. This requires your application to set the cooperative level to DSSCL_PRIORITY before setting the output format of the primary buffer.
- Create a primary sound buffer and play the buffer by using the **IDirectSoundBuffer::Play** method. This guarantees that the primary buffer is always playing, even if no secondary buffers are playing. This action consumes some of the processing bandwidth, but it reduces startup time when the first secondary buffer is played.

This section contains code samples that perform the following common tasks related to the DirectSound component. An explanation of the code accompanies each sample.

- *Creating a DirectSound Object*
- *Creating a DirectSound Object by Using CoCreateInstance*
- *Querying the Hardware Capabilities*
- *Creating Sound Buffers*
- *Writing to Sound Buffers*
- *Using the DirectSound Mixer*
- *Using a Custom Mixer*
- *Using Compressed Wave Formats*

Creating a DirectSound Object

The easiest way for your application to create a DirectSound object is to call the **DirectSoundCreate** function and specify a NULL GUID. The function will then attempt to create the object corresponding to the wave device of the default window. You must then call the **IDirectSound::SetCooperativeLevel** method; no sound buffers can be played until this call has been made. The following example demonstrates this process:

```
LPDIRECTSOUND lpDirectSound;
if(DS_OK == DirectSoundCreate(NULL, &lpDirectSound,
    NULL)) {
    // Creation succeeded.
    lpDirectSound->lpVtbl->SetCooperativeLevel(lpDirectSound,
```

```
        hwnd, DSSCL_NORMAL);
    // .
    // . Place code to access DirectSound object here.
    // .
} else {
    // Creation failed.
    // .
    // .
    // .
}
```

Your application can use the **DirectSoundEnumerate** function to specify the particular sound device to create. To use this function, you must create a **DSEnumCallback** function and, in most cases, an instance data structure, as shown in the following example:

```
typedef struct {
    // Storage for GUIDs.
    // Storage for device description strings.
} APPINSTANCEDATA, *LPAPPINSTANCEDATA;
BOOL AppEnumCallbackFunction(
    LPGUID lpGuid,
    LPTSTR lpstrDescription,
    LPTSTR lpstrModule,
    LPVOID lpContext)
{
    LPAPPINSTANCEDATA lpInstance = (LPAPPINSTANCEDATA)
    lpContext;
    // Copy GUID into lpInstance structure.
    // Strcpy description string into lpInstance
    // structure.
    return TRUE; // Continue enumerating.
}
```

Then, your application could create the DirectSound object by using the following example:

```
AppInitDirectSound()
{
    APPINSTANCEDATA AppInstanceData;
    LPGUID lpGuid;
    LPDIRECTSOUND lpDirectSound;
    HRESULT hr;
    DirectSoundEnumerate(AppEnumCallbackFunction,
    &AppInstanceData);
    lpGuid = AppLetUserSelectDevice(&AppInstanceData);

    // The application should check the return value of
    // DirectSoundCreate for errors.
```

```
    hr = DirectSoundCreate(lpGuid, &lpDirectSound, NULL);
    // .
    // .
    // .
}
```

The **DirectSoundCreate** function fails if there is no sound device or if the sound device, as specified by the *lpGuid* parameter, has been allocated through the waveform-audio 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 using the sound device.

Creating a DirectSound Object by Using CoCreateInstance

Use the following steps to create an instance of a DirectSound object by using **CoCreateInstance**:

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, &IID_IDirectSound, &lpds);
       if(!FAILED(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 the DirectSound object, you must call **IDirectSound::Initialize**. 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 **DirectSoundCreate**.

Before you close the application, shut down COM by using **CoUninitialize**, as follows:

```
CoUnitialize();
```

Querying the Hardware Capabilities

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

To retrieve the hardware capabilities, use the **IDirectSound::GetCaps** method, which fills in a **DSCAPS** structure, as shown in the following example :

```
AppDetermineHardwareCaps(LPDIRECTSOUND lpDirectSound)
{
    DSCAPS dscaps;
    HRESULT hr;
    dscaps.dwSize = sizeof(DSCAPS);
    hr = lpDirectSound->lpVtbl->GetCaps(lpDirectSound,
    &dscaps);
    if(DS_OK == hr) {
        // Succeeded, now parse DSCAPS structure.
        // .
        // .
        // .
    }
    // .
    // .
    // .
}
```

The **DSCAPS** structure contains information about the performance and resources of the sound device, including the maximum resources of each type and the resources that are currently available. There can be tradeoffs between various resources; for example, allocating a single hardware streaming sound buffer might consume two static mixing channels. 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 for the next buffer creation.

Do not make assumptions about the behavior of the sound device; otherwise, your application might work on some sound devices but not on others. Furthermore, future devices might behave differently from existing devices.

When allocating hardware resources, your application should attempt to allocate them as software buffers instead. Complete access to all hardware resources is not always available. For example, because Windows is a multitasking operating system, the **IDirectSound::GetCaps** method might indicate a free resource, but by the time you attempt to allocate the resource, it might have been allocated to another application.

Creating Sound Buffers

This section describes how your application can create simple sound buffers. It also describes the control options your application can set for each sound buffer it creates. In addition, you will find information about the differences between audio data storage in static and streaming sound buffers, hardware and software sound buffers, and in primary and secondary sound buffers.

- *Creating a Basic Sound Buffer*
- *Control Options*
- *Static and Streaming Sound Buffers*
- *Hardware and Software Sound Buffers*
- *Primary and Secondary Sound Buffers*

Creating a Basic Sound Buffer

To create a sound buffer, your application fills a **DSBUFFERDESC** structure and then calls the **IDirectSound::CreateSoundBuffer** method. This creates a DirectSoundBuffer object and returns a pointer to an *IDirectSoundBuffer* interface. Your application can use this interface to write, manipulate, and play the buffer.

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.

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;
    }
}
```

Control Options

When creating a sound buffer, your application must specify the control options needed for that buffer. This can be done with the **dwFlags** member of the **DSBUFFERDESC** structure, which can contain one or more DSBCAPS_CTRL flags. DirectSound uses the options your application specifies 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 only hardware acceleration 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 will succeed 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.

Static and Streaming Sound Buffers

A *static sound buffer* contains a complete sound in memory. These buffers are convenient because your application can write the entire sound to the buffer at once. A *streaming sound buffer* represents only a portion of a sound, such as a buffer that can hold 3 seconds of audio data that plays a 2-minute sound. In this case, your application must periodically write new data to the sound buffer. However, a streaming buffer requires much less memory than a static buffer.

Note

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.

If a sound device has onboard 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 more than once, such as the sounds of footsteps or a weapon, 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. There are no requirements for using streaming buffers. For example, you can write an entire sound to a streaming buffer if the buffer is big enough. 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 there is no need for the sound data to be downloaded to the hardware memory.

DirectSound uses the designation of a buffer as static or streaming to optimize performance; it does not restrict how you can use the buffer.

Hardware and Software Sound Buffers

A hardware sound buffer has its mixing performed by a hardware mixer located on the sound device. A software sound buffer has its mixing performed by the system central processing unit. In most cases, your application should simply specify whether the buffer is static or streaming; DirectSound locates the buffer in hardware or software as appropriate.

If your application must explicitly locate buffers in hardware or software, however, 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. Also, most existing sound devices do not have any hardware memory or mixing capacity, so no hardware buffers can be created on these devices.

You can determine the location of a sound 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.

Primary and Secondary Sound Buffers

Primary sound buffers represent the audio samples that the listener will hear. *Secondary sound buffers* each represent a single sound or stream of audio. Your application can create a primary buffer by specifying the DSBCAPS_PRIMARYBUFFER flag in the **DSBUFFERDESC** structure. If this flag is not specified, a secondary buffer is created.

Usually you should create a secondary sound buffer for every sound in an application. Note that your application can reuse sound buffers by overwriting the old sound data with new data. DirectSound takes care of the hardware resource allocation and the mixing of all buffers that are playing.

If your application uses secondary buffers, you can choose to create a primary sound buffer to perform certain controls. For example, you can control the hardware output format by calling the **IDirectSoundBuffer::SetFormat** method on the primary buffer. However, any methods that access the buffer memory, such as **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::GetCurrentPosition**, fail.

If your application performs its own mixing, DirectSound provides write access to the primary buffer. You should write data to this buffer in a timely manner; if you do not update the data, the previous buffer will repeat itself, causing gaps in the audio. Write access to the primary buffer is available only if your application sets the DSSCL_WRITEPRIMARY cooperative level. At this cooperative level, no secondary buffers can be played.

Note that your application must play primary sound buffers 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;
}
```

Writing to Sound Buffers

Your application can obtain write access to a sound buffer by using the **IDirectSoundBuffer::Lock** method. After the sound buffer (memory) is locked, the application can write or copy data to the buffer. The buffer memory must then be unlocked by calling the **IDirectSoundBuffer::Unlock** method.

Because streaming sound buffers usually play continually, 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, **IDirectSoundBuffer::Lock** would return one pointer to the last 200 bytes of the buffer, and a second pointer to the first 100 bytes. Depending on the offset and the length of the buffer, the second pointer can be NULL.

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. It can also occur when an application with the write-primary cooperative level gains the input focus. If this flag is set, DirectSound makes all other sound buffers lost so that the application with the focus can write directly to the primary buffer. If this happens, DirectSound returns the DSERR_BUFFERLOST error code in response to the **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Play** methods. When the application lowers its cooperative level from write-primary, or loses the input focus, 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.

The following example writes data to a sound buffer by using the **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Unlock** methods:

```
BOOL AppWriteDataToBuffer(
    LPDIRECTSOUNDBUFFER lpDsb, DWORD dwOffset, LPBYTE lpbSoundData,
    DWORD dwSoundBytes)
{
    LPVOID lpvPtr1;
    DWORD dwBytes1;
    LPVOID lpvPtr2;
    DWORD dwBytes2;
    HRESULT hr;
    // Obtain write pointer.
    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;
}
```

Using the DirectSound Mixer

It is easy to mix multiple streams with DirectSound. Your application can simply create secondary sound buffers, receiving an *IDirectSoundBuffer* interface for each sound. You can then use these interfaces to write data to the buffers by using the **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Unlock** methods, and you can play the buffers by using the **IDirectSoundBuffer::Play** method. You can stop playing the buffers at any time by using the **IDirectSoundBuffer::Stop** method.

The **IDirectSoundBuffer::Play** method always starts playing at the buffer's current position. The current position is specified by an offset, in bytes, into the buffer. The current position of a newly created buffer is 0. When a buffer is stopped, the current position immediately follows the next sample played. The current position can be set explicitly by calling the **IDirectSoundBuffer::SetCurrentPosition** method, and can be queried by calling the **IDirectSoundBuffer::GetCurrentPosition** method.

By default, **IDirectSoundBuffer::Play** stops playing when the end of the buffer is reached. This is the correct behavior for nonlooping static buffers. (The current position will be reset to the beginning of the buffer at this point.) For streaming buffers or for static buffers that continually repeat, your application should call **IDirectSoundBuffer::Play** and specify the DSBPLAY_LOOPING flag in the *dwFlags* parameter. This causes the buffer to loop back to the beginning once it reaches the end.

For streaming sound buffers, your application is responsible for ensuring that the next block of data is written to the buffer before the play cursor loops back to the beginning. Your application can do this by using the **SetTimer** or **SetEvent** Win32 functions to cause a message or callback function to occur at regular intervals. In addition, many DirectSound applications will already have a real-time DirectDraw component that must service the display at regular intervals; this component should be able to service DirectSound buffers as well. For optimal efficiency, all applications should write at least 1 second ahead of the current play cursor to minimize the possibility of gaps in the audio output during playback.

The DirectSound mixer can obtain the best usage 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 thereby 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; only applications with a cooperative level of DSSCL_PRIORITY or higher can call this function. DirectSound will then restore the hardware format to the format specified in the last **IDirectSoundBuffer::SetFormat** method call every time the application gains the input focus.

Using a Custom Mixer

Most applications should use the DirectSound mixer; it should be sufficient for almost all mixing needs and it automatically takes advantage of hardware acceleration, if available. 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. This feature is provided for completeness, and it should be useful only for a very limited set of high-performance applications. Applications that take advantage of this feature are subject to stringent performance requirements because it is difficult to avoid gaps in the audio.

To implement a custom mixer, the application should first obtain the DSSCL_WRITEPRIMARY cooperative level and then create a primary sound buffer. It can then call the **IDirectSoundBuffer::Lock** method, write data to the returned pointers, and then call the **IDirectSoundBuffer::Unlock** method to release the data back to DirectSound. Applications must explicitly play the primary buffer by calling the **IDirectSoundBuffer::Play** method to reproduce the sound data in the speakers. Note that the DSBPLAY_LOOPING flag must be specified or the **IDirectSoundBuffer::Play** call fails.

The following example illustrates how an application might implement a custom mixer. The AppMixIntoPrimaryBuffer function in the following example 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 we got DSERR_BUFFERLOST, 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;
}
```

Using Compressed Wave Formats

DirectSound does not currently support compressed wave formats. Applications should use the Audio Compression Manager (ACM) functions, provided with the Win32 SDK, to convert compressed audio to pulse-coded 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.

DirectSound Reference

Functions

DirectSoundCreate

```
HRESULT DirectSoundCreate(GUID FAR * lpGuid,
    LPDIRECTSOUND * ppDS, IUnknown FAR * pUnkOuter);
```

Creates and initializes an *IDirectSound* interface.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_ALLOCATED

  DSERR_INVALIDPARAM

  DSERR_NOAGGREGATION

  DSERR_NODRIVER

  DSERR_OUTOFMEMORY

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 to request 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.

The application must call the **IDirectSound::SetCooperativeLevel** method immediately after creating a DirectSound object.

See also **IDirectSound::GetCaps**, **IDirectSound::SetCooperativeLevel**

DirectSoundEnumerate

```
BOOL DirectSoundEnumerate(
    LPDSENUMCALLBACK lpDSEnumCallback, LPVOID lpContext);
```

Enumerates the DirectSound drivers installed in the system.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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.

See also **DSEnumCallback**

Callback Function

DSEnumCallback

```
BOOL DSEnumCallback(GUID FAR * lpGuid,
    LPSTR lpstrDescription, LPSTR lpstrModule,
    LPVOID lpContext);
```

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** function.

- Returns TRUE to continue enumerating drivers, or FALSE to stop.

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.

lpstrDescription
Address of a null-terminated string that provides a textual description of the DirectSound device.

lpstrModule
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.

The application can save the strings passed in the *lpstrDescription* and *lpstrModule* parameters by copying them to memory that is 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**

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. For a conceptual overview, see *IDirectSound Interface*.

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

IDirectSound::Compact

```
HRESULT Compact();
```

Moves the unused portions of onboard sound memory, if any, to a contiguous block so that the largest portion of free memory will be available.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

  DSERR_UNINITIALIZED

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.

See also *IDirectSound*, **IDirectSound::SetCooperativeLevel**

IDirectSound::CreateSoundBuffer

```
HRESULT CreateSoundBuffer(LPDSBUFFERDESC lpDSBufferDesc,
    LPLPDIRECTSOUNDBUFFER lplpDirectSoundBuffer,
    IUnknown FAR * pUnkOuter);
```

Creates a DirectSoundBuffer object to hold a sequence of audio samples.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_ALLOCATED

  DSERR_BADFORMAT

  DSERR_INVALIDPARAM

  DSERR_NOAGGREGATION

  DSERR_OUTOFMEMORY

  DSERR_UNINITIALIZED

  DSERR_UNSUPPORTED

lpDSBufferDesc
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.

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 onboard 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*, **IDirectSound::DuplicateSoundBuffer**, **IDirectSound::SetCooperativeLevel**, *IDirectSoundBuffer*, **IDirectSoundBuffer::GetFormat**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::Lock**,

IDirectSoundBuffer::Play, **IDirectSoundBuffer::SetFormat**, **IDirectSoundBuffer::SetFrequency**

IDirectSound::DuplicateSoundBuffer

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

Creates a new DirectSoundBuffer object that uses the same buffer memory as the original object.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_ALLOCATED

  DSERR_INVALIDCALL

  DSERR_INVALIDPARAM

  DSERR_OUTOFMEMORY

  DSERR_UNINITIALIZED

lpDsbOriginal
Address of the DirectSoundBuffer object to be duplicated.

lplpDsbDuplicate
Address of a pointer to the new DirectSoundBuffer object.

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.

See also *IDirectSound*, **IDirectSound::CreateSoundBuffer**

IDirectSound::GetCaps

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

Retrieves the capabilities of the hardware device that is represented by the DirectSound object.

- Returns DS_OK if successful, or one of the following error values otherwise:

DSERR_GENERIC

DSERR_INVALIDPARAM

DSERR_UNINITIALIZED

lpDSCaps

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

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 onboard 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*

IDirectSound::GetSpeakerConfig

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

Retrieves the speaker configuration specified for this DirectSound object.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_INVALIDPARAM

  DSERR_UNINITIALIZED

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 output through headphones.

DSSPEAKER_MONO

The audio is output through a single speaker.

DSSPEAKER_QUAD

The audio is output through quadraphonic speakers.

DSSPEAKER_STEREO

The audio is output through stereo speakers (default value).

DSSPEAKER_SURROUND

The audio is output through surround speakers.

See also *IDirectSound*, **IDirectSound::SetSpeakerConfig**

IDirectSound::Initialize

```
HRESULT Initialize(GUID FAR * lpGuid);
```

Initializes the DirectSound object that was created by using the **CoCreateInstance** function.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_ALREADYINITIALIZED

  DSERR_GENERIC

  DSERR_INVALIDPARAM

  DSERR_NODRIVER

lpGuid
Address of the globally unique identifier (GUID) specifying the sound driver for this DirectSound object to bind to, or NULL to select the primary sound driver.

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 that is called afterward returns **DSERR_UNINITIALIZED**.

For more information about using **IDirectSound::Initialize** with **CoCreateInstance**, see *Creating a DirectSound Object by Using CoCreateInstance*.

See also **DirectSoundCreate**

IDirectSound::SetCooperativeLevel

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

Sets the cooperative level of the application for this sound device.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_ALLOCATED

  DSERR_INVALIDPARAM

  DSERR_UNINITIALIZED

  DSERR_UNSUPPORTED

hwnd
Window handle for 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 in any application can be played.

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

IDirectSound::SetSpeakerConfig

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

Specifies the speaker configuration of the DirectSound object.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_INVALIDPARAM

  DSERR_UNINITIALIZED

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.

See also *IDirectSound*, **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 3D space. This section is a reference to the methods of this interface. For a conceptual overview, see *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

IDirectSound3DBuffer::GetAllParameters

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

Retrieves information that describes the 3D characteristics of a sound buffer at a given point in time.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

IDirectSound3DBuffer::GetConeAngles

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

Retrieves the inside and outside angles of the sound projection cone for this sound buffer.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

IDirectSound3DBuffer::GetConeOrientation

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

Retrieves the orientation of the sound projection cone for this sound buffer.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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.

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

See also **IDirectSound3DBuffer::SetConeAngles**, **IDirectSound3DBuffer::SetConeOutsideVolume**

IDirectSound3DBuffer::GetConeOutsideVolume

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

Retrieves the current cone outside volume for this sound buffer.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

Volume levels are represented by attenuation. Allowable values are between 0 (no attenuation) and -10,000 (silence). Amplification is not currently supported by DirectSound.

For additional information about the cone outside volume concept, see *Sound Projection Cones*.

See also **IDirectSoundBuffer::SetVolume**

IDirectSound3DBuffer::GetMaxDistance

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

Retrieves the current maximum distance for this sound buffer.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

By default, the maximum distance value is infinite. For additional information about minimum and maximum distances, see *Minimum and Maximum Distance Values*.

See also **IDirectSound3DBuffer::GetMinDistance**, **IDirectSound3DBuffer::SetMaxDistance**

IDirectSound3DBuffer::GetMinDistance

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

Retrieves the current minimum distance for this sound buffer.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

By default, the minimum distance value is 1.0 (corresponding to 1.0 meter at the default distance factor of 1.0 m per unit). For additional information about minimum and maximum distances, see *Minimum and Maximum Distance Values*.

See also **IDirectSound3DBuffer::SetMinDistance**, **IDirectSound3DBuffer::GetMaxDistance**

IDirectSound3DBuffer::GetMode

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

Retrieves the current operation mode for 3D sound processing.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

DS3DMODE_DISABLE
Processing of 3D 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.

IDirectSound3DBuffer::GetPosition

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

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.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

IDirectSound3DBuffer::GetVelocity

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

Retrieves the current velocity for this sound buffer.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

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

See also **IDirectSound3DBuffer::SetPosition**, **IDirectSound3DBuffer::SetVelocity**

IDirectSound3DBuffer::SetAllParameters

```
HRESULT SetAllParameters(
    LPDS3DBUFFER lpDs3dBuffer, DWORD dwApply);
```

Sets all 3D sound buffer parameters from a given **DS3DBUFFER** structure that describes all aspects of the sound buffer at a moment in time.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

lpDs3dBuffer
Address of a **DS3DBUFFER** structure containing the information that describes the 3D 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 3D coordinates for all 3D sound buffers.

IDirectSound3DBuffer::SetConeAngles

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

Sets the inside and outside angles of the sound projection cone for this sound buffer.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

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 3D coordinates for all 3D sound buffers.

Each angle must be in the range of 0 degrees (no cone) to 360 degrees (the full sphere). The default value is 360. For additional information see, *Sound Projection Cones*.

See also **IDirectSound3DBuffer::GetConeOutsideVolume**, **IDirectSound3DBuffer::SetConeOutsideVolume**

IDirectSound3DBuffer::SetConeOrientation

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

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.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for all 3D sound buffers.

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

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

Sets the current cone outside volume for this sound buffer.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

lConeOutsideVolume
Cone outside volume for this sound buffer, in hundredths of decibels. Allowable values are between 0 (no attenuation) and -10,000 (silence).

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 3D coordinates for all 3D sound buffers.

Volume levels are represented by attenuation. Amplification is not currently supported by DirectSound.

For additional information about the cone outside volume concept, see *Sound Projection Cones*.

See also **IDirectSoundBuffer::SetVolume**

IDirectSound3DBuffer::SetMaxDistance

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

Sets the current maximum distance value.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for

all 3D sound buffers.

By default, the maximum distance value is infinite. For additional information about minimum and maximum distances, see *Minimum and Maximum Distance Values*.

See also **IDirectSound3DBuffer::GetMaxDistance**, **IDirectSound3DBuffer::SetMinDistance**

IDirectSound3DBuffer::SetMinDistance

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

Sets the current minimum distance value.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for all 3D sound buffers.

By default, the minimum distance value is 1.0 (corresponding to 1.0 meter at the default distance factor of 1.0 m per unit). For additional information about minimum and maximum distances, see *Minimum and Maximum Distance Values*.

See also **IDirectSound3DBuffer::SetMaxDistance**

IDirectSound3DBuffer::SetMode

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

Sets the operation mode for 3D sound processing.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

dwMode

Flag specifying the 3D sound processing mode to be set.

DS3DMODE_DISABLE

Processing of 3D 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 3D coordinates for all 3D sound buffers.

IDirectSound3DBuffer::SetPosition

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

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.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for all 3D sound buffers.

IDirectSound3DBuffer::SetVelocity

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

Sets the sound buffer's current velocity.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for all 3D sound buffers.

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

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 3D space. This section is a reference to the methods of this interface. For a conceptual overview, see *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

IDirectSound3Dlistener ::CommitDeferredSettings

```
HRESULT CommitDeferredSettings();
```

Commits any deferred settings made since the last call to this method.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

IDirectSound3DListener::GetAllParameters

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

Retrieves information that describes the current state of the 3D world and listener.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

See also **IDirectSound3DListener::SetAllParameters**

IDirectSound3DListener::GetDistanceFactor

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

Retrieves the current distance factor.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

For additional information about distance factors, see *Distance Factor*.

See also **IDirectSound3DListener::SetDistanceFactor**

IDirectSound3DListener::GetDopplerFactor

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

Retrieves the current Doppler effect factor.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

This Doppler factor has a range of 0 (no Doppler effects) to 10.0 (10 times the Doppler effects found in the physical world). The default value is DS3D_DEFAULTDOPPLERFACTOR (1.0). For additional information about Doppler factor, see *Doppler Factor*.

See also **IDirectSound3DListener::SetDopplerFactor**

IDirectSound3DListener::GetOrientation

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

Retrieves the listener's current orientation in vectors: a front vector and a top vector.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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.

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::SetOrientation**

IDirectSound3DListener::GetPosition

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

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.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

See also **IDirectSound3DListener::SetPosition**

IDirectSound3DListener::GetRolloffFactor

```
HRESULT GetRolloffFactor(LPD3DVALUE lpflRolloffFactor);
```

Retrieves the current rolloff factor.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

lpflRolloffFactor
Address of a variable whose type is **D3DVALUE** and that will contain the current rolloff factor value.

The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0). For additional information about the rolloff factor, see *Rolloff Factor*.

See also **IDirectSound3DListener::SetRolloffFactor**

IDirectSound3DListener::GetVelocity

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

Retrieves the listener's current velocity.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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

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

```
HRESULT SetAllParameters(
    LPDS3DLISTENER lpListener, DWORD dwApply);
```

Sets all 3D listener parameters from a given **DS3DLISTENER** structure that describes all aspects of the 3D listener at a moment in time.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

lpListener
Address of a **DS3DLISTENER** structure that contains information describing all current 3D 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 3D coordinates for all 3D sound buffers.

See also **IDirectSound3DListener::GetAllParameters**

IDirectSound3DListener::SetDistanceFactor

```
HRESULT SetDistanceFactor(
    D3DVALUE flDistanceFactor, DWORD dwApply);
```

Sets the current distance factor.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for all 3D sound buffers.

For additional information about distance factors, see *Distance Factor*.

See also **IDirectSound3DListener::GetDistanceFactor**

IDirectSound3DListener::SetDopplerFactor

```
HRESULT SetDopplerFactor(
    D3DVALUE flDopplerFactor, DWORD dwApply);
```

Sets the current Doppler effect factor.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for all 3D sound buffers.

This Doppler factor has a range of 0 (no Doppler effects) to 10.0 (10 times the Doppler effects found in the physical world). The default value is DS3D_DEFAULTDOPPLERFACTOR (1.0). For additional information about Doppler factor, see *Doppler Factor*.

See also **IDirectSound3DListener::GetDopplerFactor**

IDirectSound3DListener::SetOrientation

```
HRESULT SetOrientation(D3DVALUE xFront,
    D3DVALUE yFront, D3DVALUE zFront,
    D3DVALUE xTop, D3DVALUE yTop,
    D3DVALUE zTop, DWORD    dwApply);
```

Sets the listener's current orientation in terms of two vectors: a front vector and a top vector.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

xFront, yFront, and *zFront*
Values whose types are **D3DVALUE** and that represent the coordinates of the front orientation vector.

xTop, yTop, and *zTop*
Values whose types are **D3DVALUE** and that represent the coordinates of the top orientation vector.

dwApply

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

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

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

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

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.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for all 3D sound buffers.

See also **IDirectSound3DListener::GetPosition**

IDirectSound3DListener::SetRolloffFactor

```
HRESULT SetRolloffFactor(
    D3DVALUE flRolloffFactor, DWORD dwApply);
```

Sets the rolloff factor.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for all 3D sound buffers.

The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0). For additional information about the rolloff factor, see *Rolloff Factor*.

See also **IDirectSound3DListener::GetRolloffFactor**

IDirectSound3DListener::SetVelocity

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

Sets the listener's velocity.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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 3D coordinates for all 3D sound buffers.

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 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

IDirectSoundBuffer::GetCaps

```
HRESULT GetCaps(LPDSBCAPS lpDSBufferCaps);
```

Retrieves the capabilities of the DirectSoundBuffer object.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

lpDSBufferCaps
Address of a **DSBCAPS** structure to contain the capabilities of this sound buffer.

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

```
HRESULT GetCurrentPosition(LPDWORD lpdwCurrentPlayCursor,
    LPDWORD lpdwCurrentWriteCursor);
```

Retrieves the current position of the play and write cursors in the sound buffer.

- Returns DS_OK if successful, or one of the following error values otherwise:

DSERR_INVALIDPARAM

DSERR_PRIOLEVELNEEDED

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.

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.

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.

It is always safe to change data that is behind the position indicated by the *lpdwCurrentPlayCursor* parameter.

See also *IDirectSoundBuffer*, **IDirectSoundBuffer::SetCurrentPosition**

IDirectSoundBuffer::GetFormat

```
HRESULT GetFormat(LPWAVEFORMATEX lpwfxFormat,
    DWORD dwSizeAllocated, LPDWORD lpdwSizeWritten);
```

Retrieves a description of the format of the sound data in the buffer, or the buffer size needed to retrieve the format description.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

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.

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.

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

```
HRESULT GetFrequency(LPDWORD lpdwFrequency);
```

Retrieves the frequency, in samples per second, at which the buffer is playing.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_CONTROLUNAVAIL

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

lpdwFrequency
Address of the variable that represents the frequency at which the audio buffer is being played.

The frequency value will be in the range of 100 to 100,000.

See also *IDirectSoundBuffer*, **IDirectSoundBuffer::SetFrequency**

IDirectSoundBuffer::GetPan

```
HRESULT GetPan(LPLONG lplPan);
```

Retrieves a variable that represents the relative volume between the left and right audio channels.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_CONTROLUNAVAIL

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

lplPan
Address of a variable to contain the relative mix between the left and right speakers.

The returned value is measured in hundredths of a decibel (dB), in the range of -10,000 to 10,000. The value -10,000 means the right channel is attenuated by 100 dB. The value 10,000 means the left channel is attenuated by 100 dB. The neutral value is 0; a 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 0, 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 -10,000 means that the right channel is silent and the sound is "all the way to the left," while a pan of 10,000 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

```
HRESULT GetStatus(LPDWORD lpdwStatus);
```

Retrieves the current status of the sound buffer.

- Returns DS_OK if successful, or **DSERR_INVALIDPARAM** otherwise.

lpdwStatus
Address of a variable to contain the status of the sound buffer. The status can be set to the following values:

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.

See also *IDirectSoundBuffer*

IDirectSoundBuffer::GetVolume

```
HRESULT GetVolume(LPLONG lplVolume);
```

Retrieves the current volume for this sound buffer.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_CONTROLUNAVAIL
  DSERR_INVALIDPARAM

DSERR_PRIOLEVELNEEDED

lplVolume
Address of the variable to contain the volume associated with the specified DirectSound buffer.

The volume is specified in hundredths of decibels (dB), and ranges from 0 to -10,000. The value 0 represents the original, unadjusted volume of the stream. The value -10,000 indicates an audio volume attenuated by 100 dB, which, for all practical purposes, is silence. Amplification is not currently supported by DirectSound.

The decibel scale corresponds to the logarithmic hearing characteristics of the ear. For example, an attenuation of 10 dB makes a buffer sound half as loud, and an attenuation of 20 dB makes a buffer sound one-quarter as loud.

See also *IDirectSoundBuffer*, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::Initialize

```
HRESULT Initialize(LPDIRECTSOUND lpDirectSound,
    LPDSBUFFERDESC lpDSBufferDesc);
```

Initializes a DirectSoundBuffer object if it has not yet been initialized.

- Returns **DSERR_ALREADYINITIALIZED**.

lpDirectSound
Address of the DirectSound object associated with this DirectSoundBuffer object.

lpDSBufferDesc
Address of a **DSBUFFERDESC** structure that contains the values used to initialize this sound buffer.

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

```
HRESULT Lock(DWORD dwWriteCursor, DWORD dwWriteBytes,
    LPVOID lplpvAudioPtr1, LPDWORD lpdwAudioBytes1,
    LPVOID lplpvAudioPtr2, LPDWORD lpdwAudioBytes2,
    DWORD dwFlags);
```

Obtains a valid write pointer to the sound buffer's audio data.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_BUFFERLOST

  DSERR_INVALIDCALL

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

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 flag is defined:

DSBLOCK_FROMWRITECURSOR

Locks from the current write cursor, making a call to **IDirectSoundBuffer::GetCurrentPosition** unnecessary. If this flag is specified, the *dwWriteCursor* parameter is ignored. This flag is optional.

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.

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 onboard 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 onboard memory.

See also *IDirectSoundBuffer*, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Unlock**

IDirectSoundBuffer::Play

```
HRESULT Play(DWORD dwReserved1, DWORD dwReserved2,
    DWORD dwFlags);
```

Causes the sound buffer to play from the current position.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_BUFFERLOST

  DSERR_INVALIDCALL

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

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.

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

Note

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 that define playback characteristics are superseded by the flags defined in the most recent call.

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.

Before this method can be called on any sound buffer, the application must 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 the **DSERR_PRIOLEVELNEEDED** error value.

See also *IDirectSoundBuffer*, **IDirectSound::SetCooperativeLevel**

IDirectSoundBuffer::Restore

```
HRESULT Restore();
```

Restores the memory allocation for a lost sound buffer for the specified DirectSoundBuffer object.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_BUFFERLOST

  DSERR_INVALIDCALL

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

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

```
HRESULT SetCurrentPosition(DWORD dwNewPosition);
```

Moves the current play cursor for secondary sound buffers.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_INVALIDCALL

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

dwNewPosition
New position, in bytes, from the beginning of the buffer that will be used when the sound buffer is played.

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

```
HRESULT SetFormat(LPWAVEFORMATEX lpfxFormat);
```

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.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_BADFORMAT

  DSERR_INVALIDCALL

  DSERR_INVALIDPARAM

  DSERR_OUTOFMEMORY

  DSERR_PRIOLEVELNEEDED

  DSERR_UNSUPPORTED

lpfxFormat
Address of a **WAVEFORMATEX** structure that describes the new format for the primary sound buffer.

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.

A call to this method fails if the hardware does not directly support the requested pulse coded modulation (PCM) format. It will also fail if the calling application has the DSSCL_NORMAL cooperative level.

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

```
HRESULT SetFrequency(DWORD dwFrequency);
```

Sets the frequency at which the audio samples are played.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_CONTROLUNAVAIL

  DSERR_GENERIC

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

dwFrequency

New frequency, in hertz (Hz), at which to play the audio samples. The value must be between 100 and 100,000.

If the value is 0, the frequency is reset to the current buffer format. This format is specified in the **IDirectSound::CreateSoundBuffer** method.

Increasing or decreasing the frequency changes the perceived pitch of the audio data. This method does not affect the format of the buffer.

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

IDirectSoundBuffer::SetPan

```
HRESULT SetPan(LONG lPan);
```

Specifies the relative volume between the left and right channels.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_CONTROLUNAVAIL

  DSERR_GENERIC

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

lPan

Relative volume between the left and right channels. This value has a range of -10,000 to 10,000 and is measured in hundredths of a decibel (dB).

The neutral value for *lPan* is 0; it indicates that both channels are at full volume (attenuated by 0 decibels). At any other setting, one of the channels is at full volume and the other is attenuated. For example, 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 -10,000 means that the right channel is silent and the sound is "all the way to the left," while a pan of 10,000 means that the left channel is silent and the sound is "all the way to the right."

The pan control is cumulative with the volume control.

See also *IDirectSoundBuffer*, **IDirectSoundBuffer::GetPan**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::SetVolume

```
HRESULT SetVolume(LONG lVolume);
```

Changes the volume of a sound buffer.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_CONTROLUNAVAIL

  DSERR_GENERIC

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

lVolume
New volume requested for this sound buffer. Values range from 0 (0 decibels (dB), no volume adjustment) to -10,000 (-100 dB, essentially silent). DirectSound does not currently support amplification.

Volume units of are in hundredths of decibels, where 0 is the original volume of the stream.

Positive decibels correspond to amplification and negative decibels correspond to attenuation. The decibel scale corresponds to the logarithmic hearing characteristics of the ear. An attenuation of 10 dB makes a buffer sound half as loud; an attenuation of 20 dB makes a buffer sound one-quarter as loud. DirectSound does not currently support amplification.

The pan control is cumulative with the volume control.

See also *IDirectSoundBuffer*, **IDirectSoundBuffer::GetPan**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetPan**

IDirectSoundBuffer::Stop

```
HRESULT Stop();
```

Causes the sound buffer to stop playing.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

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

```
HRESULT Unlock(LPVOID lpvAudioPtr1, DWORD dwAudioBytes1,
    LPVOID lpvAudioPtr2, DWORD dwAudioBytes2);
```

Releases a locked sound buffer.

- Returns DS_OK if successful, or one of the following error values otherwise:

  DSERR_INVALIDCALL

  DSERR_INVALIDPARAM

  DSERR_PRIOLEVELNEEDED

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.

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**

Structures

DS3DBUFFER

```
typedef struct {
    DWORD     dwSize;
    D3DVECTOR  vPosition;
    D3DVECTOR  vVelocity;
    DWORD     dwInsideConeAngle;
    DWORD     dwOutsideConeAngle;
    D3DVECTOR  vConeOrientation;
    LONG      lConeOutsideVolume;
    D3DVALUE     flMinDistance;
    D3DVALUE     flMaxDistance;
    DWORD     dwMode;
} DS3DBUFFER;
```

Contains all information necessary to uniquely describe the location, orientation, and motion of a 3D sound buffer. This structure is used with the **IDirectSound3DBuffer::GetAllParameters** and **IDirectSound3DBuffer::SetAllParameters** methods.

dwSize
Size of this structure, in bytes.

vPosition
A **D3DVECTOR** structure that describes the current position of the 3D sound buffer.

vVelocity
A **D3DVECTOR** structure that describes the current velocity of the 3D 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 3D buffer's sound projection cone.

lConeOutsideVolume
The cone outside volume.

flMinDistance
The minimum distance.

flMaxDistance
The maximum distance.

dwMode
The 3D sound processing mode to be set.

DS3DMODE_DISABLE

3D 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

```
typedef struct {
    DWORD      dwSize;
    D3DVECTOR  vPosition;
    D3DVECTOR  vVelocity;
    D3DVECTOR  vOrientFront;
    D3DVECTOR  vOrientTop;
    D3DVALUE   flDistanceFactor;
    D3DVALUE   flRolloffFactor;
    D3DVALUE   flDopplerFactor;
} DS3DLISTENER;
```

Contains all information necessary to uniquely describe the 3D world parameters and position of the listener. This structure is used with the **IDirectSound3DListener::GetAllParameters** and **IDirectSound3DListener::SetAllParameters** methods.

dwSize
Size of this structure, in bytes.

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

```
typedef struct _DSBCAPS {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwBufferBytes;
    DWORD dwUnlockTransferRate;
    DWORD dwPlayCpuOverhead;
} DSBCAPS, *LPDSBCAPS;
```

Specifies the capabilities of a DirectSound buffer object, for use by the **IDirectSoundBuffer::GetCaps** method.

dwSize
Size of this structure, in bytes.

dwFlags
Flags that specify buffer-object capabilities.

DSBCAPS_CTRL3D

The buffer is a primary buffer that uses 3D control.

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

Forces the buffer to use hardware mixing, even if DSBCAPS_STATIC is not specified. If the device does not support hardware mixing, or the required hardware memory is not available, the call to **IDirectSound::CreateSoundBuffer** 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_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 (ActiveMovie™), when the user wants to hear the soundtrack while typing in Word or 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.

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 that 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**

DSBUFFERDESC

```
typedef struct _DSBUFFERDESC{
    DWORD           dwSize;
    DWORD           dwFlags;
    DWORD           dwBufferBytes;
    DWORD           dwReserved;
    LPWAVEFORMATEX  lpwfxFormat;
} DSBUFFERDESC, *LPDSBUFFERDESC;
```

Describes the necessary characteristics of a new DirectSoundBuffer object. This structure is used by the **IDirectSound::CreateSoundBuffer** method.

dwSize
Size of this structure, in bytes.

dwFlags
Identifies the capabilities to include when creating a new DirectSoundBuffer object. Specify one or more of the following:

DSBCAPS_CTRL3D

The buffer is a primary buffer that uses 3D 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_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_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 (ActiveMovie), when the user wants to hear the soundtrack while typing in Word or 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.

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.

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

```
typedef struct _DSCAPS {
    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;
```

Specifies the capabilities of a DirectSound device for use by the **IDirectSound::GetCaps** method.

dwSize
Size of this structure, in bytes.

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 for this release.

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 trade-offs 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 3D positional capabilities of the device. These will all be 0 for the first release.

dwFreeHw3DAllBuffers, **dwFreeHw3DStaticBuffers**, and **dwFreeHw3DStreamingBuffers**
Description of the free, or unallocated, hardware 3D positional capabilities of the device. These will all be 0 for the first release.

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 onboard 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**
These values are reserved. Do not use.

See also **IDirectSound::GetCaps**

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_OTHERAPPHASPRIO

This value is obsolete and is not used.

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.