Microsoft® DirectX™ 2 Software Development Kit

C H A P T E R 3

DirectSound

Overview

About DirectSound

The Microsoft® DirectSound™ application programming interface (API) is the audio component of the Microsoft Windows® 95 DirectX™ 2 Software Development Kit (SDK) that provides low-latency mixing, hardware acceleration, and direct access to the sound device. DirectSound provides this functionality while maintaining compatibility with existing Windows 95-based applications and device drivers.

DirectX 2 allows you, as an application developer, access to the display and audio hardware while insulating you from the specific details of that hardware. The overriding design goal in DirectX 2 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.

Object Types

The most fundamental type of object is the DirectSound object, which represents the sound card itself. The DirectSound object is controlled by the **IDirectSound** Component Object Model (COM) interface; 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.

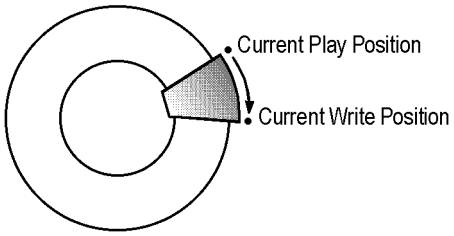
Primary buffers control sound characteristics, such as output format and total volume. Also, your application can write directly to the primary buffer. In this case, however, the DirectSound mixing and hardware acceleration features are not available. In addition, writing directly to the primary buffer may 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 used by an application. Each buffer can be played or stopped independently. DirectSound mixes all playing buffers into the primary buffer, then 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 buffers or streaming buffers. A static buffer means that the buffer contains an entire sound. A streaming buffer means that the buffer only contains part of a sound, and, therefore, your application must continually write new data to the buffer while it is playing. DirectSound will attempt to store static buffers 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 the 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 the current play and current write positions.



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

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, only producing sound 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. In this way, applications do not have to repeatedly play and stop their buffers when the input focus changes.

In a future release, it will be possible to open special audio streams that are not muted when the application loses the input focus.

Note 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. DirectSoundBuffer objects can be created by calling the **IDirectSound::CreateSoundBuffer** method that returns an **IDirectSoundBuffer** interface.

Applications are also able to create primary or secondary sound buffers. As previously stated, a secondary sound buffer represents a single sound or audio stream. A primary buffer represents the output audio stream that can be a composite of several mixed secondary 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 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. Once there, it is possible to 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. A buffer that is playing can be stopped by using the **IDirectSoundBuffer::Stop** method.

Generally, 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 buffer to store the sound. If the sound is longer than that, you should use a streaming 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. DirectSound attempts to store static buffers 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 will locate streaming buffers in hardware only if the sound device is located on a fast bus.

Note The DSBCAPS_STATIC flag used with the **IDirectSound::CreateSoundBuffer** method determines whether the buffer is static or streaming. If this flag is specified, DirectSound creates a static buffer; otherwise, DirectSound creates a streaming buffer.

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 95 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.

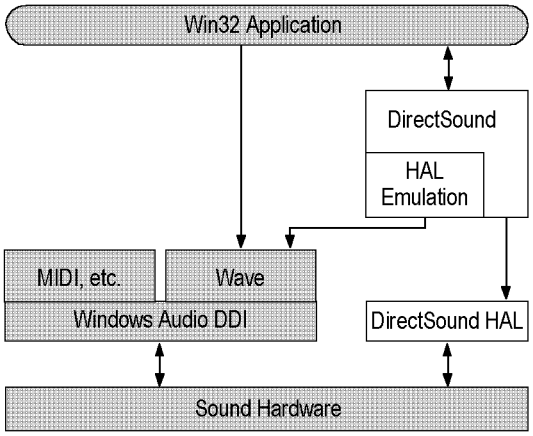
The device driver does not perform any software emulation; it simply reports the capabilities of the hardware to DirectSound and passes requests from DirectSound to the hardware. If the hardware cannot perform a requested operation, the device driver fails the request and DirectSound emulates the operation.

If a DirectSound driver is not available, DirectSound communicates with the audio hardware through the standard Windows 95 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.

System Integration

The following diagram 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 will fail the call. DirectSound then emulates the request in software.

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, trying to allocate that same device using DirectSound will fail. Similarly, if a DirectSound driver has allocated a device, trying to allocate the device using the waveform-audio driver will fail.

If your application needs to 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 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 Microsoft 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** function or the **WaveOut** function.

Note Microsoft Video For Windows currently uses the waveform-audio functions to output the audio track of an .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 the **IDirectSound::Release** method before playing an .avi file, and then re-create and reinitialize the DirectSound object and its DirectSoundBuffer objects when the video finishes playing. For more information about **IDirectSound::Release,** see **IUnknown Interface**.

DirectSound Features

Mixing

The most used feature of DirectSound is the low-latency mixing of audio streams. 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. There are no limitations to the number of buffers that can be mixed, except the practicalities of available processing time.

Low-latency mixing allows the user to experience no perceptible delay between the time that a buffer plays and the time that the speakers reproduce the sound. In practical terms, this means that 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.

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

Hardware Acceleration

DirectSound automatically takes advantage of accelerated sound hardware, including hardware mixing and hardware sound-buffer memory. Your application does not need to 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; those that access the primary buffer directly are subject to very 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 in order to prevent the previous block in the buffer from repeating. During buffer creation, you cannot specify the size of the buffer and must accept the returned size once creation is complete.

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.

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.

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 that 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.

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 output to the sound device. These samples can be a single audio stream or the result of mixing several audio streams. The audio data in a primary sound buffer is typically not accessed directly by applications. However, the primary buffer can be used for control purposes, such as setting the output volume or wave format.

Secondary sound buffers represent a single output stream or sound. Your application can play these buffers into the primary sound buffer. Secondary sound buffers that play concurrently are mixed into the primary buffer, which is then sent to the sound device.

Play Management

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

The **IDirectSoundBuffer::Lock** method retrieves a write pointer into the current sound buffer. After writing audio data into 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 are played. The frequency of the primary buffer cannot be changed.

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

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. The **IDirectSoundBuffer::GetFormat** and **IDirectSoundBuffer::SetFormat** methods can also be used to set the format of the sound data in the primary buffer.

Note Once a secondary buffer is created, its format is fixed. If a secondary buffer that uses another format is needed, you should create a new sound buffer with the necessary 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 content of the memory. Once the buffer memory is restored, it must be rewritten with valid sound data.

IUnknown Interface

Like all Component Object Model (COM) interfaces, the **IDirectSound** and **IDirectSoundBuffer** interfaces also include the **AddRef**, **Release**, and **QueryInterface** methods. The **AddRef** method increases the reference count of the object and the **Release** method decreases the reference count. Your applications can use the **QueryInterface** method to determine what additional interfaces an object supports.

When an object is created, its reference count is set to 1. Each time a new application binds to the object or a previously bound application binds to a different interface of the object, the reference count is increased by 1. Each time an application is released from an interface, the reference count is decreased by 1. The object deallocates itself when its reference count goes to 0. The **Release** method notifies the object that an application is no longer bound to the object.

Your application can use the **QueryInterface** method to ask an object if it supports a particular interface. If the interface is supported, it can be used immediately. If the interface is not needed, you must call the **Release** method to free it. **QueryInterface** allows objects to be extended by Microsoft and third parties without breaking or interfering with each other's existing or future functionality.

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

Implementation: A Broad Overview

This section describes the programming model for DirectSound and provides some guidelines for typical tasks.

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 buffers using the **IDirectSound::CreateSoundBuffer** method. Your application need not specify that they are secondary buffers in the **DSBUFFERDESC** structure; creating 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 operations:

- Set the output format of the primary buffer by creating a primary sound buffer and calling the **IDirectSoundBuffer::SetFormat** method. This operation 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 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.

Creating a DirectSound Object

The easiest way to create a DirectSound object is for your application to call the **DirectSoundCreate** function and specify a NULL GUID. The function will then attempt to create the object corresponding to the default window's wave device.

You must then call the **IDirectSound::SetCooperativeLevel** method; no sound buffers can be played until this call has been made:

```
LPDIRECTSOUND lpDirectSound;
if(DS_OK == DirectSoundCreate(NULL, &lpDirectSound,
    NULL)) {
    // Create succeeded!
    lpDirectSound->lpVtbl->SetCooperativeLevel(lpDirectSound,
        hwnd, DSSCL_NORMAL);
    //  .
    //  .  Place code to access DirectSound object here.
    //  .
} else {
    // Create failed!
    //  .
    //  .
    //  .
}
```

The **DirectSoundEnumerate** function can be used 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:

```
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, to create the DirectSound object, you would use code like this:

```
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 will fail 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 will either continue without sound or prompt the user to close the application that is using the sound device.

Creating a DirectSound Object Using CoCreateInstance

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

1 Initialize COM at the start of your application using CoInitialize(NULL).

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

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

```
dsrval = CoCreateInstance(&CLSID_DirectSound,
                                        NULL,
                                        CLSCTX_ALL,
                                        &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. *lpds* is the uninitialized object **CoCreateInstance** returns.

Before you use the DirectSound object, you must call **IDirectSound::Initialize**. **IDirectSound::Initialize** takes the driver GUID parameter that **DirectSoundCreate** typically uses (NULL in this case). Once the DirectSound

object is initialized, you can use and release the DirectSound object as if it had been created using **DirectSoundCreate**.

Before closing the application, shut down COM using **CoUninitialize**, as shown below.

```
    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 will fill in a **DSCAPS** structure:

```
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 may be trade-offs between various resources; for example, allocating a single hardware streaming 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 may work on some sound devices but not on others. Furthermore, advanced devices are under development that will behave differently than 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 95 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 may have been allocated to another application.

Creating Sound Buffers

Creating a Basic Sound Buffer

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

You should create buffers for the most important sounds first, and continue to create them 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 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);
    dsbdesc.dwFlags = DSBCAPS_CTRLDEFAULT; // Need default controls
                                           (pan, volume, frequency).
    dsbdesc.dwBufferBytes = 3 * pcmwf.wf.nAvgBytesPerSec; // 3 second
```

```
                                                                buffer.
    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 this information when allocating hardware resources to sound buffers. For example, a particular device might support hardware buffers but provide no pan control on those buffers. In this case, DirectSound would only use hardware acceleration if the DSBCAPS_CTRLPAN flag was not specified.

To obtain the best performance on all sound cards, your application should only specify those control options which it will actually 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, it 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 the entire sound can be written once to the buffer.

A streaming buffer only represents a portion of a sound, such as a buffer that can hold three seconds of audio data that plays a two-minute sound. In this case, your application must periodically write new data into the sound buffer. However, a streaming buffer requires much less memory than a static buffer.

When creating a sound buffer, you can indicate that a buffer is static by specifying the DSBCAPS_STATIC flag. If this flag is not specified, the buffer is a streaming buffer.

If a sound device has onboard sound memory, DirectSound will attempt to place static buffers in the hardware memory. These buffers can then take advantage of hardware mixing, with the benefit that the processing system incurs little or no overhead to mix these sounds. This is particularly useful for sounds that will be played more than once, such as the sound a character makes while walking or a firearm going off, since the sound data only needs to be downloaded once to the hardware memory.

Streaming buffers are generally located in main system memory to allow efficient writing to the buffer, although hardware mixing can be used on peripheral component interconnect (PCI) machines or other fast buses. There are no requirements on the use of streaming buffers. For example, you can write an entire sound into a streaming buffer if it is big enough. In fact, if you do not intend to use the sound more than once, it may be more efficient to use a streaming buffer because there is no need for the sound data to be downloaded to the hardware memory.

Note	The designation of a buffer as static or streaming is used by DirectSound 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 will locate the buffer in hardware or software as appropriate.

If explicitly locating buffers in hardware or software is necessary, however, you can specify either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags in the **DSBUFFERDESC** structure. If the DSBCAPS_LOCHARDWARE flag is specified and there is insufficient hardware memory or mixing capacity, the buffer creation request will fail. Also, most existing sound devices do not have any hardware memory or mixing capacity, so no hardware buffers can be created on these devices.

The location of a sound buffer can be determined by calling 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 will always be specified.

Primary and Secondary Sound Buffers

Primary sound buffers represent the actual audio samples that the listener will hear. Secondary 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 will be created.

Usually you should create secondary sound buffers for each sound in a given application. Note that sound buffers can be reused by overwriting the old sound data with new data. DirectSound takes care of the hardware resource allocation and the mixing together of all buffers that are playing.

If your application uses secondary buffers, you may 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 actual buffer memory, such as **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::GetCurrentPosition**, will fail.

If your application performs its own mixing, DirectSound provides write access to the primary buffer. You should write data into 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 only available if your application has the DSSCL_WRITEPRIMARY cooperative level. At this cooperative level, no secondary buffers can be played.

Note that primary sound buffers must be played with looping. Be sure 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;
    dsbdesc.dwBufferBytes = 0; // Buffer size is determined
```

```
                                // by sound hardware.
    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;
            }
        }
    }
    // If we got here, then we failed SetCooperativeLevel.
    // 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. Once the sound buffer (memory) is locked, you can write or copy data to the buffer. The buffer memory must then be unlocked by calling the **IDirectSoundBuffer::Unlock** method.

Since streaming sound buffers usually wrap around and continue playing from the beginning of the buffer, DirectSound returns two write pointers when locking a sound buffer. For example, if you try to lock 300 bytes beginning at the midpoint of a 400 byte buffer, **IDirectSoundBuffer::Lock** would return a 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 may be NULL.

You should be aware that memory for a sound buffer can be lost in certain situations. In particular, this might occur when buffers are located in the hardware sound memory. In the most dramatic case, the sound card itself might be removed from the system while being used; this situation can occur with PCMCIA sound cards. It can also occur when an application with the write-primary cooperative level (DSSCL_WRITEPRIMARY flag) gains the input focus. If this flag is set,

DirectSound makes all other sound buffers become 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 function writes data to a sound buffer 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 we got DSERR_BUFFERLOST, 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;
        }
    }
```

```
    // If we got here, then we failed Lock, Unlock, or Restore.
    return FALSE;
}
```

Using the DirectSound Mixer

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

The **IDirectSoundBuffer::Play** method will always start 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** will stop playing when it reaches the end of the buffer. This is the correct behavior for non-looping 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 continuously repeat, your application should call **IDirectSoundBuffer::Play** and specify the DSBPLAY_LOOPING flag in the *dwFlags* parameter. This will cause the buffer to loop back to the beginning once it reaches the end.

For streaming buffers, your application is responsible for ensuring that the next block of data is written into the buffer before the play cursor loops back to the beginning. This can be done by using the Win32 functions **SetTimer** or **SetEvent** 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 which needs to 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 one 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 out of 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 will download these buffers to the sound hardware memory, where available, and will thereby not incur any processing overhead in mixing these buffers. The most important static buffers should be created first, in order to give them first priority for hardware acceleration.

The DirectSound mixer will produce the best sound quality if all of 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.

The hardware output format can be changed by creating a primary 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 each 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 buffer and mix streams directly into it. This feature is provided for completeness, and should only be useful for a very limited set of high-performance applications. Applications that take advantage of this feature are subject to very stringent performance requirements, since 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 into the returned pointers, and 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 will fail.

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

```
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 we wrapped around.
        }
        // Release the data back to DirectSound.
        hr = lpDsbPrimary->lpVtbl->Unlock(lpDsbPrimary, lpvPtr1,
            dwBytes1, lpvPtr2, dwBytes2);
        if(DS_OK == hr) {
            // Success!
            return TRUE;
        }
    }
    // If we got here, then we failed Lock or Unlock.
    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 into the sound buffer for maximum efficiency.

Compressed wave formats will be supported in a future release.

Reference

Functions

The DirectSound functions create a DirectSound object, which in turn controls the direct-audio memory access enabled by the DirectX 2 SDK. DirectSound virtualizes direct memory access (DMA) to the audio buffer of the sound card. This allows the application, using the DirectSound object, to mix and play its own audio.

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, to request the default device, NULL.

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 Functions

Most of the functionality of DirectSound is provided by the methods of its Component Object Model (COM) interfaces. This section lists the callback functions that are not implemented as part of a COM interface.

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 into 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 Interface

IDirectSound Interface Method Groups

Applications use the methods of the IDirectSound interface to create DirectSound objects and set up the environment. The methods can be organized into the following groups:

Allocating memory	**Compact**
	Initialize
Creating buffers	**CreateSoundBuffer**
	DuplicateSoundBuffer
	SetCooperativeLevel
Device capabilities	**GetCaps**
IUnknown	**AddRef**
	QueryInterface
	Release
Speaker configuration	**GetSpeakerConfig**
	SetSpeakerConfig

All COM interfaces inherit the **IUnknown** interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the DirectSound object without affecting the functionality of the original interface.

IDirectSound::AddRef

```
ULONG AddRef();
```

Increases the reference count of the DirectSound object by 1. This method is part of the **IUnknown** interface inherited by DirectSound.

- Returns the new reference count of the object.

When the DirectSound object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the **IDirectSound::Release** method to decrease the reference count of the object by 1.

See also **IDirectSound::QueryInterface**, **IDirectSound::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, in order to take advantage of hardware mixing. To force the buffer to use either hardware or software mixing, use the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags.

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 since 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. This information can be used to fine-tune performance and optimize resource allocation.

Because of resource sharing requirements, the maximum capabilities in one area might only be available at the cost of another area. For example, the maximum number of hardware-mixed streaming buffers may only be available if there are no hardware static 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 if it has not yet been initialized.

- Returns **DSERR_ALREADYINITIALIZED**.

lpGuid

Address of the GUID specifying the sound driver for this DirectSound object to bind to, or NULL to select the primary sound driver.

Because the **DirectSoundCreate** function calls this method internally, it is not needed for the current release of DirectSound.

See also **DirectSoundCreate**

IDirectSound::QueryInterface

```
HRESULT QueryInterface(REFIID  riid, LPVOID * ppvObj);
```

Determines if the DirectSound object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the **IUnknown** interface inherited by DirectSound.

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

  DSERR_GENERIC **DSERR_INVALIDPARAM**

riid

Reference identifier of the interface being requested.

ppvObj

Address of a location to be filled with the returned interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirectSound::QueryInterface** method allows DirectSound objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

See also **IDirectSound::AddRef**, **IDirectSound::Release**

IDirectSound::Release

```
ULONG Release();
```

Decreases the reference count of the DirectSound object by 1. This method is part of the **IUnknown** interface inherited by DirectSound.

- Returns the new reference count of the object.

The DirectSound object deallocates itself when its reference count reaches 0. Use the **IDirectSound::AddRef** method to increase the reference count of the object by 1.

See also **IDirectSound::AddRef**, **IDirectSound::QueryInterface**

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. 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.

DSSCL_NORMAL

Sets the application to a fully cooperative status. Most applications should use this level, since it has the smoothest multi-tasking 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 buffers. No secondary 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.

Four cooperative levels are defined: normal, priority, exclusive, and 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, monaural 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, when it has the input focus, only this application's buffers are audible. Once the input focus is gained, DirectSound restores the application's preferred wave format, which was defined in the most recent call to **IDirectSoundBuffer::SetFormat**.

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 using the **IDirectSoundBuffer::Lock** method and write directly into 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 using the **IDirectSoundBuffer::Restore** method before they can be played again. When the application then loses the input focus, its primary buffer is marked as lost and can be restored after the application regains the input focus.

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

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 for 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**

IDirectSoundBuffer Interface

IDirectSoundBuffer Interface Method Groups

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
IUnknown	**AddRef**
	QueryInterface
	Release
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, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the DirectSoundBuffer object without affecting the functionality of the original interface.

IDirectSoundBuffer::AddRef

```
ULONG AddRef();
```

Increases the reference count of the DirectSoundBuffer object by 1. This method is part of the **IUnknown** interface inherited by DirectSound.

- Returns the new reference count of the object.

When the DirectSoundBuffer object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the **IDirectSoundBuffer::Release** method to decrease the reference count of the object by 1.

See also **IDirectSoundBuffer::QueryInterface**, **IDirectSoundBuffer::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 location of the buffer, 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 into 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 being played. This value will be in the range of 100-100,000.

- 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.

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. 0 is the neutral value; a pan value of 0 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 one of 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 being played. 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 (dB) 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 into the pointers returned by the **IDirectSoundBuffer::Lock** method and then call the **IDirectSoundBuffer::Unlock** method to release the buffer back to DirectSound. The sound buffer should not be locked for long periods of time; if it is, the play cursor will reach the locked bytes and configuration-dependent audio problems, possibly random noise, will result.

Warning This method returns a write pointer only. The application should not try to read sound data from this pointer; the data may not be valid even though the DirectSoundBuffer object contains valid sound data. For example, if the buffer is located in onboard memory, the pointer may 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 buffers.

For secondary buffers, this method will cause the buffer to be mixed into the primary buffer and output to the sound device. If this is the first buffer to play, it will implicitly create a primary buffer and start playing that buffer; the application does not need to 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.

For primary sound buffers, this method will cause them 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 output 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 may reduce processing overhead when sounds are started and stopped in sequence since the primary buffer will be playing continuously rather than stopping and starting between secondary buffers.

Note Before this method can be called on any sound buffer, the application 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::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID * ppvObj);
```

Determines if the DirectSoundBuffer object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the **IUnknown** interface inherited by DirectSound.

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

  DSERR_INVALIDPARAM **DSERR_GENERIC**

riid
Reference identifier of the interface being requested.

ppvObj
Address of a location that will be filled with the returned interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The

IDirectSoundBuffer::QueryInterface method allows DirectSoundBuffer objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

See also **IDirectSoundBuffer::AddRef**, **IDirectSoundBuffer::Release**

IDirectSoundBuffer::Release

```
ULONG Release();
```

Decreases the reference count of the DirectSoundBuffer object by 1. This method is part of the **IUnknown** interface inherited by DirectSound.

- Returns the new reference count of the object.

The DirectSound object deallocates itself when its reference count reaches 0. Use the **IDirectSoundBuffer::AddRef** method to increase the reference count of the object by 1.

See also **IDirectSoundBuffer::AddRef**, **IDirectSoundBuffer::QueryInterface**

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** may 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 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.

A call to this method fails if the sound buffer is playing or 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 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 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.

0 is the neutral value for *lPan* and 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 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 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 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 only play 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 dB.

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 into 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 into 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

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_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_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 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 will be muted, but the sticky focus buffers will still be audible. This is useful for non-game applications, such as movie playback (ActiveMovie™) when the user wants to hear the soundtrack while typing in Word or Excel, for example. If the user switches to DirectSound, all sound buffers, both normal and sticky focus, in the previous application will be 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 since no processing is required. For hardware buffers, the rate may be slower because the buffer might have to be downloaded to the sound card, which may 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_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_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 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.

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 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 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 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.

dwMaxHwMixingStaticBuffers
Specifies the maximum number of static buffers.

dwMaxHwMixingStreamingBuffers
Specifies the maximum number of streaming buffers.

Note The value for **MaxHwMixingAllBuffers** may be less than the sum of **dwMaxHwMixingStaticBuffers** and **dwMaxHwMixingStreamingBuffers**. Resource trade-offs frequently occur.

---

dwFreeHwMixingAllBuffers, **dwFreeHwMixingStaticBuffers**, and **dwFreeHwMixingStreamingBuffers**
Description of the free, or unallocated, hardware mixing capabilities of the device. These values can be used by an application 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.

dwUnlockTranferRateHwBuffers
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 does not need to be transferred anywhere. Similarly, the play processing overhead for hardware buffers is 0 because the mixing is done by the sound device.

dwReserved1, **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 is capable of returning, 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.