DirectSound

DirectSound

DirectSound

DirectSound

DirectSound

DirectSound

About DirectSound

Microsoft® DirectSound™application programming interface (API) is the audio component of the Microsoft Windows® 95 Game 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.

The Windows Game subsystem allows game developers access to the display and audio hardware while insulating them from the specific details of that hardware. The overriding design goal in the Windows Game subsystem 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

Object Types

The most fundamental kind of DirectSound object is controlled by the **IDirectSound** component object model (COM) interface; this object represents the sound card itself. Member functions of this interface allow your 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. Secondary sound buffers represent individual source sounds. DirectSound provides controls for primary and secondary sound buffers in the **IDirectSoundBuffer** COM interface.

Primary buffers control sound characteristics, such as output format and total volume. An application can also write directly to the primary buffer. However, in this case, the DirectSound mixing and hardware acceleration features will not be available. In addition, applications that write directly to the primary buffer may interfere with other DirectSound applications. When possible, applications 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, and they allow the application to share the sound card with other applications in the system.

Secondary buffers represent a single sound source used by the 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 CPU overhead.

Secondary sound buffers can be either streaming buffers or static buffers. A *static buffer* means that the buffer contains the entire sound. A *streaming buffer* means that the buffer only contains part of the sound, and therefore the 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 CPU time when they are played, because the mixing is done in hardware. Reusable sounds, such as gunshots, are the perfect candidates for static buffers.

Your application 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 spot in the buffer at which the sound is being played. The current write position indicates the spot at which your application can safely change the data in the buffer.

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. Applications can create a DirectSound object by calling the **DirectSoundCreate** function that returns an **IDirectSound** interface. Applications can enumerate DirectSound objects installed in the system by calling the **DirectSoundEnumerate** function.

Windows is a multitasking operating system. Typically users run several programs at once, and expect all of them to cooperate and share resources. DirectSound objects share sound devices by tracking the input focus and only produce sound when the owning application currently 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 these applications, the audio output automatically switches from the streams of one application to the streams of the other. Applications do not have to play and stop their buffers when the input focus changes.

Note

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

The DirectSoundBuffer Object

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

Applications can 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 one and only one primary buffer.

Applications can write data into sound buffers by locking the buffer, writing data to the buffer, and unlocking the buffer. Your application can lock the buffer by using the **IDirectSoundBuffer::Lock** member function. This member function returns a pointer to the locked portion of the buffer. You can copy audio data to the buffer . After writing data to the buffer, your application must unlock the buffer and complete the write operation by using the **IDirectSoundBuffer::Unlock** member function.

The primary sound buffer contains data that is heard. Your application can play audio data from a secondary sound buffer by using the **IDirectSoundBuffer::Play** member function. **IDirectSoundBuffer::Play** 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. Your application can also play a sound repeatedly in a continuous loop by specifying the DSBPLAY_LOOPING flag when it calls this member function. Your application can also stop a buffer that is playing by using the **IDirectSoundBuffer::Stop** member function.

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

Your application can create a DirectSoundBuffer object that has a static buffer by using the **IDirectSound::CreateSoundBuffer** member function 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 the sound hardware do not consume system CPU time when they are played, because the mixing is done in hardware. Reusable sounds (such as engine roars, cheers, and jeers) are perfect candidates for static buffers.

Streaming buffers can also use hardware mixing if the sound device supports this; however, this is efficient when an application runs on computers with fast data buses, such as the 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** member function 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 emulates the features that a particular sound card does not directly support in software without any loss of functionality . Applications can query DirectSound to determine the capabilities of the audio hardware by using the **IDirectSound::GetCaps** member function. A high-performance game could 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 driver is a Windows 95 audio-device driver that has been modified to support the HAL. This driver architecture provides backwards 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

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.

An 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 functions.

The DirectSound functions and the waveform audio functions provide alternative paths to the waveform-audio portion of the sound hardware. A single device is limited to access from one path at a time. If a waveform driver has allocated a device, trying to allocate the device using DirectSound will fail. Similarly, if a DirectSound driver has allocated a device, trying to allocate the device using the waveform driver will fail.

If an application needs to use both sets of functions, use each set sequentially. For example, your application 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** member function. 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, an 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** functions.

Note

Microsoft Video For Windows currently uses the waveform audio functions to output the audio track of an AVI file. Therefore, if an application is using DirectSound and it plays an AVI file, the audio track will not be audible. Similarly, if the application is playing an AVI file and it attempts to create a DirectSound object, the creation function will return an error.

For now, an application can release the DirectSound object by calling the **IDirectSound::Release** member function before playing an AVI file, and then recreate 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
- Hardware Acceleration
- Access to the Primary Buffer

Mixing

The most used feature of DirectSound is low-latency mixing of audio streams. An application creates one or more secondary sound buffers and writes audio data to them. An application can 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 is no limit to the number of buffers that can be mixed (except the practical limitations of available CPU time).

Low-latency means the user experiences no perceptible delay between the time that a buffer plays and the time that the speakers reproduce the sound. In practical terms, this means the latency is 20 milliseconds or less. DirectSound mixer provides 20 milliseconds of latency and play begins with no perceptible delay. Therefore, if an application plays a buffer and immediately begins a screen animation, the audio and the video appear to start synchronously. Note, however, that if DirectSound uses the HAL emulation layer (that is, 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 that application are audible.

Hardware Acceleration

DirectSound automatically takes advantage of accelerated sound hardware, including hardware mixing and hardware sound-buffer memory. Applications do not need to query the hardware or program specifically to use hardware acceleration.

However, if an application wants to make the best possible use of the hardware resources, it can query DirectSound to receive a full description of the hardware capabilities of the sound device. From this information, the application can specify which sound buffers should receive hardware acceleration.

Because the application determines when to use each effect, when to play each buffer, and the priority of each sound buffer, it can allocate hardware resources as it needs them.

Access to the Primary Buffer

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

A primary buffer is typically very small, so an application that writes directly to one of these buffers must write blocks of data at short intervals to prevent the previous block in the buffer from being repeated. An application cannot specify the size of the buffer and must simply accept the buffer size returned when the buffer is created.

When an application obtains 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 applications should use secondary buffers instead of directly accessing the primary buffer. An application can write to a secondary buffer easily because its larger buffer size provides much more time to write the next block of data, thereby minimizing the risk of gaps in the audio. Even an application that has simple audio requirements, such as using one stream of audio data where mixing is not needed, 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 Introduction. The **IDirectSound** COM interface enables an application to define and control the sound card, speaker, and memory environment.

This topic discusses the member functions of the **IDirectSound** COM interface.

Device Capabilities

After calling **DirectSoundCreate** to create a DirectSound object, your application can retrieve the capabilities of the sound device by using the **IDirectSound::GetCaps** member function. For optimal performance, your application should call **IDirectSound::GetCaps** to determine the capabilities of the resident sound card and modify the game's 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** member function creates a sound buffer.
IDirectSound::DuplicateSoundBuffer creates a second sound buffer using the same physical buffer memory. An application that duplicates a sound buffer can play both buffers independently without wasting buffer memory.

Your application must use the **IDirectSound::SetCooperativeLevel** member function to set its cooperation level for a sound device before playing any sound buffers. Most applications use a “normal” priority level, which ensures that they will not conflict with other applications.

Speaker Configuration

The **IDirectSound** interface contains two member functions that allow your application to investigate and set the configuration of the system's speakers. These member functions 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** member function to move any onboard sound memory into a contiguous block to make the largest portion of free memory available.

IDirectSoundBuffer Introduction

Audio data resides in a DirectSound buffer. An 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.

The **IDirectSoundBuffer** COM interface enables your application to work with buffers of audio data. This topic discusses the member functions of the **IDirectSoundBuffer** interface.

Play Management

Your application can use the **IDirectSoundBuffer::Play** and **IDirectSoundBuffer::Stop** member functions to control the real-time playback of sound. Your application can play a sound using **IDirectSoundBuffer::Play**. If the buffer is played with looping specified, the buffer repeats until your application calls **IDirectSoundBuffer::Stop**. Otherwise, the buffer stops automatically when the end of the buffer is reached.

The **IDirectSound::Lock** member function retrieves a write pointer into the current sound buffer. After writing audio data into the buffer, your application must unlock the buffer by using the **IDirectSound::Unlock** member function. Applications should not leave the buffer locked for extended periods

To retrieve or set the current position in the sound buffer, an application can call **IDirectSound::GetCurrentPosition** or **IDirectSound::SetCurrentPosition**.

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** member functions. Setting the volume on the primary sound buffer has the effect of changing the waveform volume of the sound card.

Similarly, the **IDirectSoundBuffer::GetFrequency** and **IDirectSoundBuffer::SetFrequency** functions 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, your application can call the **IDirectSoundBuffer::GetPan** and **IDirectSoundBuffer::SetPan** member functions. The pan of the primary buffer cannot be changed.

Information

The **IDirectSoundBuffer::GetCaps** member function retrieves the capabilities of the DirectSoundBuffer object. Your application can use the **IDirectSoundBuffer::GetStatus** member function to find out whether the current sound buffer is playing or has stopped.

Your application can use the **IDirectSoundBuffer::GetFormat** member function to retrieve information about the format of the sound data in the buffer. Your application can use the **IDirectSoundBuffer::GetFormat** and **IDirectSoundBuffer::SetFormat** member functions to set the format of the sound data in the primary buffer.

Note

The format for a secondary buffer is fixed. If your application needs a secondary buffer that uses another format, it should create a new sound buffer with the desired format.

Memory Management

Your application can use the **IDirectSoundBuffer::Restore** memory-management function to restore the sound buffer memory for a specified DirectSoundBuffer object. Although this is useful if the buffer has been lost, the **IDirectSoundBuffer::Restore** function cannot restore the content of the memory, only the memory itself. Once the buffer memory is restored, it must be rewritten with valid sound data.

IUnknown Interface

Like all COM interfaces, the **IDirectSound** and **IDirectSoundBuffer** interfaces also include the **AddRef**, **Release,** and **QueryInterface** member functions. The **AddRef** function increases the reference count of the object and the **Release** function decreases the reference count. Applications use the **QueryInterface** function to determine what additional interfaces are supported by an object. For a description of these functions, see DirectDraw.

When an object is created, its reference count is set to one. Each time a new application binds to the object or a previously bound application binds to a different COM interface of the object, the reference count is increased by one. The object deallocates itself when its reference count goes to zero. The **Release** member notifies the object that an application is no longer bound to the object.

An application can use the **QueryInterface** function to ask an object if it supports a particular COM interface. If it does, the application may begin using that interface immediately. If the application does not want to use that interface, it must call **Release** 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.

The **AddRef** function uses one parameter that points to the object: either **LPDIRECTSOUND** or **LPDIRECTSOUNDBUFFER**. This function returns a doubleword value specifying the new reference count for the object.

The **Release** function uses one parameter that points to the object: either **LPDIRECTSOUND** or **LPDIRECTSOUNDBUFFER**. This function returns a doubleword value specifying the new reference count for the object.

The **QueryInterface** function uses two parameters in addition to the first parameter that points to the object. The second parameter, whose type is **REFIID**, specifies a GUID identifying the requested interface. The third parameter is a pointer to a location that is filled with the returned interface pointer if the query is successful; the type of this parameter is **LPVOID FAR ***. **QueryInterface** returns an HRESULT: DS_OK if it is successful, or either DSERR_INVALIDPARAM or E_NOINTERFACE if there is an error.

Note

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

Using DirectSound

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

Implementation: A Broad Overview

Your application should follow these basic steps to implement DirectSound. These steps and additional functionality are discussed in greater detail later in this <chapter>.

1. Create a DirectSound object, by calling the **DirectSoundCreate** function.
2. Specify a cooperative level, by calling **IDirectSound::SetCooperativeLevel**. Most applications use the lowest level, DSSCL_NORMAL.
3. Create secondary buffers, using **IDirectSound::CreateSoundBuffer**. You need not specify that they are secondary buffers in the **DSBUFFERDESC** structure, because creating secondary buffers is the default case.
4. Load the secondary buffers with data. Use **IDirectSoundBuffer::Lock** to obtain a pointer to the data area and **IDirectSoundBuffer::Unlock** to set the data to the device.
5. Use **IDirectSoundBuffer::Play** to play the secondary buffers.
6. Stop all buffers when you have finished playing sounds, using **IDirectSoundBuffer::Stop** method of the DirectSoundBuffer object.
7. Release the secondary buffers.
8. Release the DirectSound object.

Your application can also perform the following optional items:

1. Set the output format of the primary buffer by creating a primary sound buffer and calling **IDirectSoundBuffer::SetFormat**. This operation requires your application to set the cooperative level to DSSCL_PRIORITY before setting the output format of the primary buffer.
2. Create a primary sound buffer and play the buffer using **IDirectSoundBuffer::Play**. This guarantees that the primary buffer is always playing, even if no secondary buffers are playing. This action consumes some of the CPU 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 to call **DirectSoundCreate** and specify a NULL GUID. This will attempt to create the object corresponding to the default Windows wave device. You must then call **IDirectSound::SetCooperativeLevel**; 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 **DirectSoundEnumCallback** function, and usually 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, the application 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 wave functions. Applications should be prepared for this call to fail, and should either continue without sound or prompt the user to close the application that is using the sound device.

Querying the Hardware Capabilities

DirectSound provides the ability to retrieve the hardware capabilities of the sound device used by a DirectSound object. Most applications will not need to do this; DirectSound will automatically take advantage of hardware acceleration without intervention by the application. However, high-performance applications can use this information to scale their sound requirements to the available hardware. For example, an application might play more sounds if hardware mixing is available.

To retrieve the hardware capabilities, use the **IDirectSound::GetCaps** member function, 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 currently free resources. Note that there may be trade-offs between the various resources; for example, allocating a single hardware streaming buffer might consume two static mixing channels. Applications which scale to the hardware capabilities should call **IDirectSound::GetCaps** between every buffer allocation to determine if there are enough resources for the next buffer, rather than making assumptions about the resource trade-offs.

Do not make assumptions about the behavior of the sound device; otherwise your application will work on some sound devices but not others. Furthermore, advanced devices are currently under development which will behave differently from existing devices.

When attempting to allocate hardware resources, always handle the error case gracefully (that is, attempt to allocate it as a software buffer instead). Applications should not assume that they have complete access to all hardware resources. Because Windows 95 is a multitasking operating system, the **IDirectSound::GetCaps** member function might indicate a free resource, but by the time the application attempts to allocate the resource, it may have been allocated to another application.

Creating Sound Buffers

Creating Sound Buffers

This topic discusses the topics such as creating a basic sound buffer and distinguishing between primary and secondary buffers.

Creating a Basic Sound Buffer

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

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

The following code fragment 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, applications must specify the control options needed for that buffer. The dwFlags member of the **DSBUFFERDESC** structure 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 not provide pan control on those buffers. In this case, DirectSound would only use the hardware acceleration if DSBCAPS_CTRLPAN was not specified.

To obtain the best performance on all sound cards, applications should only specify those control options which they will actually use.

If an application attempts to call a control function that a buffer lacks, the function fails. For example, if an application attempts to change the volume by using **IDirectSoundBuffer::SetVolume**, the function succeeds if the DSBCAPS_CTRLVOLUME flag was specified when the buffer was created. Otherwise, the function fails and returns the DSERR_CONTROLUNAVAIL code. Providing controls for the buffers helps to ensure that all applications run correctly on all sound devices, present and future.

Static and Streaming Sound Buffers

A static sound buffer is a buffer that contains a complete sound in memory. Static buffers are convenient because the entire sound can be written once to the buffer.

A streaming buffer is a buffer that only represents a portion of the sound, such as a buffer that can hold three seconds of audio data that plays a two-minute sound. In this case, the application must periodically write new data into the sound buffer; however, the buffer takes up much less memory.

When creating a sound buffer, an application 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.

On sound devices with onboard sound memory, DirectSound attempts to locate static buffers in the hardware memory. These buffers can then take advantage of hardware mixing with the benefit that the system CPU incurs 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) because 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 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 reuse the sound (that is, play it more than once) it may be more efficient to use a streaming buffer because the sound data need not be downloaded to the hardware memory.

Note

The designation of a buffer as static versus streaming is used by DirectSound to optimize performance; it does not restrict how an application can use the buffer.

Hardware and Software Sound Buffers

A hardware sound buffer is a buffer whose mixing is performed by a hardware mixer located on the sound device. A software sound buffer is a buffer whose mixing is performed by the system CPU. In most cases, applications should simply specify whether the buffer is a static or streaming buffer; DirectSound will locate the buffer in hardware or software as appropriate.

However, if an application wishes to explicitly locate buffers in hardware or software, it may do so by specifying either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags in the **DSBUFFERDESC** structure. Note that if the DSBCAPS_LOCHARDWARE flag is specified and there is insufficient hardware memory or mixing capacity, the buffer creation request will fail. Note also that 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 (hardware or software) can be determined by calling **IDirectSoundBuffer::GetCaps** 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

A primary sound buffer represents the actual audio samples that the listener will hear. A secondary buffer represents a single sound or stream of audio. An 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.

Most applications should create secondary sound buffers for each sound in the application (or each portion of the application; sound buffers can be reused by overwriting the sound data). DirectSound takes care of hardware resource allocation, and mixing together all playing buffers.

Applications which use secondary buffers may want to create a primary sound buffer to perform certain control functions. For example, an application can control the hardware output format by calling **IDirectSoundBuffer::SetFormat** on the primary buffer. However, any functions that access the actual buffer memory (such as **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::GetCurrentPosition**) will fail.

For applications that perform their own mixing, DirectSound provides write access to the primary buffer. The application is responsible for writing data into this buffer in a timely manner; if the application neglects to update the data, the previous buffer will repeat itself causing gaps in the audio. Write access to the primary buffer is only available to applications which have 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 (that is, the DSBPLAY_LOOPING flag must be set).

The following code fragment is an example of 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

An application can obtain write access to a sound buffer by using the **IDirectSoundBuffer::Lock** member function. Once locked, an application can write or copy data to the buffer. An application must unlock the buffer memory by using the **IDirectSoundBuffer::Unlock** member function.

When locking a sound buffer, DirectSound returns two write pointers. This is because streaming sound buffers usually wrap around and continue playing from the beginning of the buffer. For example, if an application tries to lock 300 bytes starting halfway into 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.

Developers 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 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. DirectSound makes all other sound buffers lost so that the application with the focus can write directly to the primary buffer. If this happens, DirectSound returns the DSERR_BUFFERLOST error code in response to **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Play**. 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 **IDirectSoundBuffer::Restore**. If successful, this function restores the buffer memory and all other setting for the buffer, such as volume and pan settings. However, a restored buffer does not contain valid sound data. The application must rewrite the data to the restored buffer.

The following function writes data to a sound buffer using **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Unlock**:

```
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. An application simply creates secondary buffers, receiving an IDirectSoundBuffer interface for each sound. It then uses these interfaces to write data into the buffers using the **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Unlock** member functions and plays the buffers using the **IDirectSoundBuffer::Play** member function. The buffers can be stopped at any time by calling the **IDirectSoundBuffer::Stop** member function.

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

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

For streaming buffers, the 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 timeSetEvent to cause a message or callback 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, an application 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 the 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 CPU overhead in mixing these buffers. The most important static buffers should be created first, in order to give them first priority at hardware acceleration.

The DirectSound mixer will produce the best sound quality if all of the application's sounds use the same wave format and the application matches the hardware output format to the format of the sounds. Then the mixer does not need to perform any format conversion.

The hardware output format can be changed by creating a primary buffer and calling **IDirectSoundBuffer::SetFormat**. 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** call whenever the application gains the input focus.

Using a Custom Mixer

Most applications should use the DirectSound mixer, as it should be sufficient for almost all mixing needs and it automatically takes advantage of hardware acceleration, when 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 this buffer. 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; it is difficult to avoid gaps in the audio.

To implement a custom mixer, an application should first obtain the DSSCL_WRITEPRIMARY cooperative level and then create a primary sound buffer. The application can then call **IDirectSoundBuffer::Lock**, write data into the returned pointers, and call **IDirectSoundBuffer::Unlock** to release the data back to DirectSound. The application must explicitly play the primary buffer by calling **IDirectSoundBuffer::Play** to reproduce the sound data at 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 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.

Functions

 DirectSoundCreate
 DirectSoundEnumCallback
 DirectSoundEnumerate

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 values otherwise:

DSERR_ALLOCATED	The function failed because resources (such as a priority level) were already in use by another caller.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_NOAGGREGATION	This object does not support aggregation.
DSERR_NODRIVER	No sound driver is available for use.
DSERR_OUTOFMEMORY	The DirectSound subsystem couldn't allocate sufficient memory to complete the caller's request.

lpGuid

Address of the GUID identifying the sound device. The value of this parameter must be one of the GUIDs returned by DirectSoundEnumerate or NULL to request the default device.

ppDS

Address of a pointer to a DirectSound object created in response to this member function.

pUnkOuter

Controlling unknown of the aggregate. Must be NULL.

An application must call **IDirectSound::SetCooperativeLevel** immediately after creating a DirectSound object

DirectSoundEnumCallback

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

Application-defined callback function to enumerate the DirectSound drivers. The system calls this function in response to an application's previous call to the DirectSoundEnumerate function.

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

lpGuid

The GUID identifying the DirectSound driver being enumerated. This value can be passed to DirectSoundCreate to create a DirectSound object for that driver.

lpstrDescription

Address of a null-terminated string giving a textual description of the DirectSound device.

lpstrModule

Address of a null-terminated string specifying the module name of the DirectSound driver corresponding to this device.

lpContext

Address of application-defined data that is passed to each callback function.

An application can save the strings passed in *lpstrDescription* and *lpstrModule*, by copying them into memory allocated from the heap. The memory used to pass the strings to this callback function is valid only during this callback function.

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 **DirectSoundEnumCallback** callback 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 it is made.

IDirectSound Interface and Member Functions

IDirectSound

Designates an interface that enables an application to define and control the sound card, speaker, and memory environment. The member functions of the IDirectSound interface can be broken down into the following functional groups.

Creating buffers	
CreateSoundBuffer	Creates a DirectSoundBuffer object that holds a sequence of audio samples.
DuplicateSoundBuffer	Creates a new DirectSoundBuffer object that uses the same buffer memory as the original object.
SetCooperativeLevel	Sets the cooperative level of this application for this sound device.
Allocating memory	
Compact	Moves all of the pieces of onboard sound memory to a contiguous block to make the largest portion of free memory available.
Initialize	Initializes a DirectSound object.
Device capabilities	
GetCaps	Retrieves the capabilities of the hardware device represented by the DirectSound object.
Speaker configuration	
GetSpeakerConfig	Returns the speaker configuration specified for this DirectSound object.
SetSpeakerConfig	Specifies the speaker configuration for this DirectSound object.

Like all COM interfaces, **IDirectSound** also includes the **QueryInterface**, **AddRef**, and **Release** functions. For a description of these functions, see DirectDraw.

IDirectSound::Compact

```
HRESULT Compact(LPDIRECTSOUND lpDirectSound);
```

Moves the unused portions of onboard sound memory (if any) to a contiguous block to make the largest portion of free memory available.

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

DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSound

Address of the DirectSound object to compact.

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

IDirectSound::CreateSoundBuffer

```
HRESULT CreateSoundBuffer(LPDIRECTSOUND lpDirectSound,
    LPDSBUFFERDESC lpDSBufferDesc,
    LPDIRECTSOUNDBUFFER * lplpDirectSoundBuffer,
    IUnknown FAR * pUnkOuter)
```

Creates a DirectSoundBuffer object for holding a sequence of audio samples.

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

DSERR_ALLOCATED	The function failed because resources (such as a priority level) were already in use by another caller.
DSERR_BADFORMAT	The specified wave format is not supported.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_NOAGGREGATION	This object does not support aggregation.
DSERR_OUTOFMEMORY	The DirectSound subsystem couldn't allocate sufficient memory to complete the caller's request.

lpDirectSound
Address of the DirectSound object to associate with the new DirectSoundBuffer object.

lpDSBufferDesc
Address of a **DSBUFFERDESC** structure that contains a description of the sound buffer to create.

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

pUnkOuter
Controlling unknown of the aggregate. Must be NULL.

An application must specify a cooperative level for a DirectSound object by using the **IDirectSound::SetCooperativeLevel** member function before it can play any sound buffers.

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

You can also specify the DSBCAPS_STATIC flag, in which case DirectSound attempts to store the buffer in onboard memory (if available), to take advantage of hardware mixing. To force the buffer to use either hardware or software mixing, use the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags.

DirectSound supports two types of sound buffers: primary and secondary. A primary buffer represents the final audio data that will be output by the sound device and heard by the listener. A secondary buffer represents a single channel of audio, before effects or mixing has been applied. In order to use most of the mixing and hardware acceleration features of DirectSound, applications must use secondary sound buffers. The **CreateSoundBuffer** function creates secondary sound buffers by default. Samples in a secondary sound buffer must be played (often by using **IDirectSoundBuffer::Play**) to be audible to the listener; the buffer will then be mixed into the primary buffer (whether or not the application has a pointer to the primary) and output to the sound device.

Any application may create a primary buffer, which will allow it to call such member functions as **IDirectSoundBuffer::GetFormat**, **IDirectSoundBuffer::SetFormat**, and **IDirectSoundBuffer::GetVolume**. To create a primary buffer, specify the DSBCAPS_PRIMARYBUFFER flag in the **DSBUFFERDESC** structure. In addition, to get access to the audio samples themselves, the application must have write-primary cooperation, which is obtained by calling **IDirectSound::SetCooperativeLevel** and requesting the DSSCL_WRITEPRIMARY level. If the application is not in this mode, then all calls to **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Play** will fail.

For secondary buffers, an application must specify a wave format in the **DSBUFFERDESC** structure. This must be a pulse coded modulation (PCM) format.

For primary buffers, , an application must specify a NULL waveform format and set the **dwBufferBytes** member of the **DSBUFFERDESC** structure to zero. Using this value for **dwBufferBytes** creates a primary buffer of optimal size for the sound device. Once a primary buffer is created, an application can retrieve the format and size of the primary buffer by using the **IDirectSoundBuffer::SetFormat** member function. An application can also change the format by using **IDirectSoundBuffer::SetFormat**.

IDirectSound::DuplicateSoundBuffer

```
HRESULT DuplicateSoundBuffer(LPDIRECTSOUND lpDirectSound,
    LPDIRECTSOUNDBUFFER lpDsbOriginal,
    LPDIRECTSOUNDBUFFER* 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 values otherwise:

DSERR_ALLOCATED	The function failed because resources (such as a priority level) were already in use by another caller.
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_OUTOFMEMORY	The DirectSound subsystem couldn't allocate sufficient memory to complete the caller's request.

lpDirectSound

Address of the DirectSound object to associate with the new DirectSoundBuffer object.

lpDsbOriginal

Address of the DirectSoundBuffer object to duplicate.

lplpDsbDuplicate

Address to contain 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, an application can change the parameters of the buffers independently; both buffers can be played or stopped independently of one another.

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.

IDirectSound::GetCaps

```
HRESULT GetCaps(LPDIRECTSOUND lpDirectSound,LPDSCAPS lpDSCaps);
```

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

- Returns DS_OK if successful or DSERR_INVALIDPARAM otherwise.

lpDirectSound

Address of the DirectSound object to examine.

lpDSCaps

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

Information retrieved in the **DSCAPS** structure contains information about the maximum capabilities of the sound device and currently available capabilities, such as the number of hardware mixing channels and amount of onboard sound memory. This information can be used to tune performance and optimize resource allocation.

Due to 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.

An application that explicitly requests hardware resources should retrieve an accurate description of the sound card capabilities by calling **GetCaps** before creating each buffer, and should create buffers in order of their importance.

DirectSound can emulate all hardware functions using the system CPU and memory.

IDirectSound::GetSpeakerConfig

```
HRESULT GetSpeakerConfig(LPDIRECTSOUND lpDirectSound,
    LPDWORD lpdwSpeakerConfig);
```

Retrieves the speaker configuration specified for this DirectSound object.

- Returns DS_OK if successful or DSERR_INVALIDPARAM otherwise.

lpDirectSound

Address of the DirectSound object with the speaker configuration to examine.

lpdwSpeakerConfig

Address to contain the speaker configuration for this DirectSound object. The speaker configuration is indicated with one of the following values:

DSSPEAKER_HEADPHONE	Audio is output through headphones.
DSSPEAKER_MONO	Audio is output through a single speaker.
DSSPEAKER_QUAD	Audio is output through quadraphonic speakers.
DSSPEAKER_STEREO	Audio is output through stereo speakers (default value).
DSSPEAKER_SURROUND	Audio is output through surround speakers.

IDirectSound::Initialize

```
HRESULT Initialize(LPDIRECTSOUND lpDirectSound, GUID FAR *lpGuid)
```

Initializes the DirectSound object if it has not yet been initialized.

- Returns DSERR_ALREADYINITIALIZED.

lpDirectSound

Address of the DirectSound object for which this function is being called.

lpGuid

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

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

IDirectSound::SetCooperativeLevel

```
HRESULT SetCooperativeLevel(LPDIRECTSOUND lpDirectSound, HWND hwnd,
    DWORD dwLevel)
```

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

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

DSERR_ALLOCATED	The function failed because resources (such as a priority level) were already in use by another caller.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.

lpDirectSound

Address of the DirectSound object for which this function is being called.

hwnd

Window handle of this application.

dwLevel

Requested priority level. Specify one of the following values:

DSSCL_EXCLUSIVE	Sets this application to exclusive level. When this application has the input focus, it will be the only application audible (even in future releases of DirectSound). This application also has all the privileges of the DSSCL_PRIORITY level. DirectSound will restore the hardware format (as specified by the most recent call to **IDirectSoundBuffer::SetFormat**) when the application gets the input focus.
DSSCL_NORMAL	Sets the application to fully cooperative status. Most applications should use this level, as it is guaranteed to have the smoothest multitasking and resource-sharing behavior.
DSSCL_PRIORITY	Sets this application to priority level. Applications with this cooperative level can call **IDirectSoundBuffer::SetFormat** and **IDirectSound::Compact**.
DSSCL_WRITEPRIMARY	The highest priority level. The application has write access to the primary buffers. No secondary buffers (in this application or other applications) can be played.

An application must set the cooperative level by calling this member function 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**, IDirectSoundBuffer::SetFormat** and **IDirectSound::Compact** cannot be called; in addition, the application cannot get 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 and, in addition, only that application's buffers are audible when the application has the input focus. When the application gains the input focus, DirectSound restores the application's preferred wave format (defined in the most recent call to **IDirectSoundBuffer::SetFormat**).

When using a DirectSound object with the write-primary cooperative level, the application has direct access to the primary sound buffer. In this mode, the application must lock the buffer using **IDirectSoundBuffer::Lock** and write directly into the primary buffer; secondary buffers cannot be played.

When an application with the write-primary cooperative level gains the input focus, all secondary buffers for other applications are stopped, marked as lost, and must be restored using **IDirectSoundBuffer::Restore** before they can be played. When an application with the write-primary cooperative level loses the input focus, its primary buffer is marked as lost, and can be restored after the application regains the input focus.

Note that the write-primary level is not required in order to create a primary buffer. However, to get 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 functions such as **IDirectSoundBuffer::GetFormat**, **IDirectSoundBuffer::SetFormat**, and **IDirectSoundBuffer::GetVolume** can still be called.

IDirectSound::SetSpeakerConfig

```
HRESULT SetSpeakerConfig(LPDIRECTSOUND lpDirectSound,
    DWORD dwSpeakerConfig);
```

Specifies the speaker configuration for this DirectSound object.

- Returns DS_OK if successful or DSERR_INVALIDPARAM otherwise.

lpDirectSound

Address of the DirectSound object for which this function is being called.

dwSpeakerConfig

Speaker configuration of the specified DirectSound object. Specify one of the following values:

DSSPEAKER_HEADPHONE	Speakers are headphones.
DSSPEAKER_MONO	Speakers are monaural.
DSSPEAKER_QUAD	Speakers are quadraphonic.
DSSPEAKER_STEREO	Speakers are stereo (default value).
DSSPEAKER_SURROUND	Speakers are surround sound.

IDirectSoundBuffer Interface and Member Functions

IDirectSoundBuffer

Designates an interface that enables an application to work with audio data in memory. The member functions of the **IDirectSoundBuffer** interface can be broken down into the following functional groups.

Play management	
GetCurrentPosition	Retrieves the current position of the sound playing cursor in the sound buffer.
Lock	Obtains a valid pointer to the sound buffer's audio data.
Play	Causes the sound buffer to start playing.
SetCurrentPosition	Moves the current play cursor for a DirectSoundBuffer object.
Stop	Causes the sound buffer to stop playing.
Unlock	Releases a locked sound buffer.
Sound-environment management	
GetFrequency	Retrieves the frequency at which the buffer is being played.
GetPan	Retrieves a variable that represents the mix between the left and right speakers.
GetVolume	Retrieves the current volume for this sound buffer.
SetFrequency	Sets the frequency at which the audio samples are to be played.
SetPan	Specifies the mix between the left and right speakers.
SetVolume	Changes the volume of a sound buffer.
Information	
GetCaps	Retrieves the capabilities of the DirectSoundBuffer object.
GetFormat	Retrieves a **DSBUFFERFORMAT** structure with a detailed description of the format of the sound data in this sound buffer.
GetStatus	Retrieves a variable containing the current status of the sound buffer.
SetFormat	Changes the format of a sound buffer to the format described in the **DSBUFFERFORMAT** structure.
Memory management	
Initialize	Initializes a DirectSoundBuffer object, if it has not yet been initialized.
Restore	Restores the sound buffer memory for the specified DirectSoundBuffer object, if possible.

Like all COM interfaces, **IDirectSoundBuffer** also includes the **QueryInterface**, **AddRef**, and **Release** member functions. For a description of these functions, see DirectDraw.

IDirectSoundBuffer::GetCaps

```
HRESULT GetCaps(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    LPDSBCAPS lpDSBufferCaps);
```

Retrieves the capabilities of the DirectSoundBuffer object.

- Returns DS_OK if successful or DSERR_INVALIDPARAM otherwise.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to examine.

lpDSBufferCaps

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

The **DSBCAPS** structure contains similar information to the **DSBUFFERDESC** structure passed to **CreateSoundBuffer**, with some additional information. This additional information can include the location of the buffer (hardware or software), and some cost measures. Examples of cost measures include the time to download to a hardware buffer and CPU overhead to mix and play the buffer when the buffer is in 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 are used to force the buffer to be located in either hardware or software.

IDirectSoundBuffer::GetCurrentPosition

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

Retrieves the current position of the sound-playing and sound-writing cursors in the sound buffer.

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

DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to examine.

lpdwCurrentPlayCursor

Address of a variable to contain the current playing 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 writing position in the DirectSoundBuffer object. This position is an offset within the sound buffer and is specified in bytes.

The write cursor is the position at which it is safe to write new data into the buffer. The write cursor always leads the play cursor and is typically spaced ahead in the buffer 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.

IDirectSoundBuffer::GetFormat

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

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

- Returns DS_OK if successful or DSERR_INVALIDPARAM otherwise.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to examine.

lpwfxFormat

Address of the **WAVEFORMATEX** structure to contain a description of the sound data in this sound buffer. Specify NULL to retrieve the buffer size needed to contain the format description.

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, an application should query the DirectSoundBuffer object for the size of the format by calling this member function 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.

IDirectSoundBuffer::GetFrequency

```
HRESULT GetFrequency(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    LPDWORD lpdwFrequency);
```

Retrieves the frequency at which the buffer is being played, in samples per second. This value will be in the range of 100-100,000.

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

DSERR_CONTROLUNAVAIL	The control (volume, pan, and so forth) requested by the caller is not available.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object for which this function is being called.

lpdwFrequency

Address of the variable for the frequency at which the audio buffer is being played.

IDirectSoundBuffer::GetPan

```
HRESULT GetPan(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    LPLONG lplPan);
```

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

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

DSERR_CONTROLUNAVAIL	The control (volume, pan, and so forth) requested by the caller is not available.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to examine.

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. Zero is the neutral value; a pan of zero means that both channels are at full volume (that is, they are attenuated by zero decibels). At any setting other than zero, 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 (the sound is “all the way to the left”), and a pan of 10,000 means that the left channel is silent (the sound is “all the way to the right”).

The pan control is cumulative with volume control.

IDirectSoundBuffer::GetStatus

```
HRESULT GetStatus(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    LPDWORD lpdwStatus);
```

Retrieves the current status of the sound buffer.

- Returns DS_OK if successful or DSERR_INVALIDPARAM otherwise.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to examine.

lpdwStatus

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

DSBSTATUS_BUFFERLOST	The buffer is lost, and must be restored before it can be played or locked.
DSBSTATUS_LOOPING	Buffer is being looped. If this value is not set, the buffer will stop when it reaches the end. Note that if this value is set, the buffer must also be playing.
DSBSTATUS_PLAYING	Buffer is being played. If this value is not set, the buffer is stopped.

IDirectSoundBuffer::GetVolume

```
HRESULT GetVolume(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    LPLONG lplVolume);
```

Retrieves the current volume for this sound buffer.

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

DSERR_CONTROLUNAVAIL	The control (volume, pan, and so forth) requested by the caller is not available.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to examine.

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 zero to -10,000. The value zero represents the original, unadjusted volume of the stream. The value -10000 indicates an audio volume attenuated by -100 db, which is essentially silence. Amplification is not currently supported.

The decibel (dB) scale corresponds to the logarithmic hearing characteristics of the ear: positive decibels. An attenuation of 20 dB makes a buffer sound half as loud; an attenuation of 40 dB makes a buffer sound one quarter as loud.

IDirectSoundBuffer::Initialize

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

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

- Returns DSERR_ALREADYINITIALIZED.

lpDirectSoundBuffer

Address of the DirectSound object to initialize.

lpDirectSound

Address of the DirectSound object associated with this DirectSoundBuffer object.

lpDSBufferDesc

Address of a **DSBUFFERDESC** structure that containing the values to use to initialize this sound buffer.

Because the **IDirectSound::CreateSoundBuffer** function calls **IDirectSoundBuffer::Initialize** internally, it is not needed for the current release of DirectSound. This function is provided for future extensibility.

IDirectSoundBuffer::Lock

```
HRESULT Lock(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    DWORD dwWriteCursor, DWORD dwWriteBytes,
    LPLPVOID lplpvAudioPtr1, LPDWORD lpdwAudioBytes1,
    LPLPVOID 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 values otherwise:

DSERR_BUFFERLOST	The buffer memory has been lost and must be restored.
DSERR_INVALIDCALL	Indicates the buffer is already locked and has not been unlocked.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to lock.

dwWriteCursor

Offset, in bytes, from the start of the buffer 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. The sound buffer is circular.

lplpvAudioPtr1

Address of a pointer to contain the first block of the sound buffer that is 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 a second block of the sound buffer that is 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 zero.

dwFlags

Flags modifying the lock event. DSBLOCK_FROMWRITECURSOR is optional. It locks from the current write cursor, making a call to **IDirectSoundBuffer::GetCurrentPosition** unnecessary. If this flag is specified, the *dwWriteCursor* parameter is ignored.

This member function accepts an offset and a count of bytes, 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.

Applications should write data into the pointers returned by **IDirectSoundBuffer::Lock** and then call **IDirectSoundBuffer::Unlock** to release the buffer back to DirectSound. The sound buffer should not be locked for long periods of time; otherwise the play cursor will reach the locked bytes, and configuration-dependent audio problems (possibly random noise) will result.

Warning

This function returns a write pointer only. Applications should not try to access sound data by reading from this

pointer, as 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 point to a temporary buffer in main system memory; when **IDirectSoundBuffer::Unlock** is called, this temporary buffer will be transferred to the onboard memory.

IDirectSoundBuffer::Play

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

Causes the sound buffer to start playing from the current position.

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

DSERR_BUFFERLOST	The buffer memory has been lost and must be restored.
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_PRIOLEVELNEEDED	Cooperative level has not been set.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to play. The audio is copied into the primary sound buffer and played.

dwReserved1

Reserved; must be zero.

dwReserved2

Reserved; must be zero.

dwFlags

Flags specifying how to play the buffer. The following flag is defined:

DSBPLAY_LOOPING	Play restarts at the beginning of the audio buffer when the end of the buffer is reached. Play continues until it is explicitly stopped. This flag must always be set when playing primary buffers.

For secondary buffers, this member function 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 that buffer playing; the application does not need to explicitly tell the primary buffer to play.

If the buffer specified in the member function is already playing, the call to the member function will succeed and the buffer continues to play. However, the flags that define playback characteristics are superseded by the flags defined in the most recent call.

Primary buffers must always be played with the DSBPLAY_LOOPING flag set.

For primary buffers, this member function will cause the primary sound buffer to start playing to the sound device. If the application has 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 has any other cooperative level, this application will simply ensure that the primary buffer is playing even when no secondaries are playing (it will play silence in that case). This may reduce overhead when sounds are started and stopped in sequence, because the primary buffer will play continuously rather than stopping and starting between secondary buffers.

Note

Before this member function can be called on any sound buffer, the application must call **IDirectSound::SetCooperativeLevel** and specify a cooperative level (typically DSSCL_NORMAL). If **IDirectSoundBuffer::SetCooperativeLevel** has not been called, the **IDirectSoundBuffer::Play** function returns DSERR_PRIOLEVELNEEDED.

IDirectSoundBuffer::Restore

```
HRESULT Restore(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer);
```

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

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

DSERR_BUFFERLOST	The buffer memory could not be restored.
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_PRIOLEVELNEEDED	Cooperative level has not been set.

lpDirectSoundBuffer

Address of the DirectSound object to have its memory allocation restored.

If the application does not have input focus, **IDirectSoundBuffer::Restore** might not succeed. For example, if the input focus application has the DSSCL_WRITEPRIMARY cooperative level, no other application will be able to restore its buffers. Similarly, an application with DSSCL_WRITEPRIMARY cooperative level can only restore the primary buffer when it has the input focus.

Once DirectSound restores the buffer memory, an application must rewrite the buffer with valid sound data. DirectSound cannot restore the contents of the memory, only the memory itself.

An application can receive notification that a buffer is lost when it specifies that buffer in a call to the **IDirectSoundBuffer::Lock** or **IDirectSoundBuffer::Play** member function. These member functions return DSERR_BUFFERLOST to indicate a lost buffer. The **IDirectSoundBuffer::GetStatus** can also be used to retrieve the status of the sound buffer and test for the DSBSTATUS_BUFFERLOST flag.

IDirectSoundBuffer::SetCurrentPosition

```
HRESULT SetCurrentPosition(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    DWORD dwNewPosition);
```

Moves the current play cursor for secondary sound buffers.

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

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_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object whose cursor is moved.

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 begin playing from the new position. If it is not playing, it will begin playing from the new position the next time **IDirectSoundBuffer::Play** is called.

IDirectSoundBuffer::SetFormat

```
HRESULT SetFormat(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    LPWAVEFORMATEX lpwfxFormat);
```

Sets the format of the primary sound buffer for this 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 values otherwise:

DSERR_BADFORMAT	The specified wave format is not supported.
DSERR_INVALIDCALL	The specified buffer is a secondary buffer.
DSERR_INVALIDPARAM	An invalid parameter was passed to a returning function.
DSERR_OUTOFMEMORY	The DirectSound subsystem could not allocate sufficient memory to complete the request.
DSERR_PRIOLEVELNEEDED	Cooperative level has not been set.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to recieve the format change.

lpfxFormat

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

A call to this member function fails if the sound buffer is playing or the hardware does not directly support the requested PCM format. It will also fail if the calling application has DSSCL_NORMAL cooperative level.

If a secondary buffer requires a format change, an application should create a new DirectSoundBuffer object using the new format.

DirectSound supports PCM formats; it does not currently support compressed formats.

IDirectSoundBuffer::SetFrequency

```
HRESULT SetFrequency(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    DWORD dwFrequency);
```

Sets the frequency at which the audio samples are played.

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

DSERR_CONTROLUNAVAIL	The control (volume, pan, and so forth) requested by the caller is not available.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to receive the playback frequency change.

dwFrequency

New frequency, in Hz, to play audio samples (by calling the **IDirectSoundBuffer::Play** function). The value must be between 100 and 100,000.

If the value is zero, the frequency is reset to the current buffer format (specified in **IDirectSound::CreateSoundBuffer).**

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

IDirectSoundBuffer::SetPan

```
HRESULT SetPan(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer, LONG lPan);
```

Specifies the relative volume between the left and right channels.

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

DSERR_CONTROLUNAVAIL	The control (volume, pan, and so forth) requested by the caller is not available.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer to receive the new pan setting.

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.

Zero is the neutral value for *lPan* and indicates that both channels are at full volume (that is, they are attenuated by zero 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 (the sound is “all the way to the left”), and a pan of 10,000 means that the left channel is silent (the sound is “all the way to the right”).

The pan control is cumulative with volume control.

IDirectSoundBuffer::SetVolume

```
HRESULT SetVolume(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer,
    LONG lVolume);
```

Changes the volume of a sound buffer.

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

DSERR_CONTROLUNAVAIL	The control (volume, pan, and so forth) requested by the caller is not available.
DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to receive the new volume setting.

lVolume

New volume requested for this sound buffer. Values range from 0 (0 db, no volume adjustment) to -10,000 (-100 db, essentially silent). Amplification is not currently supported.

Volume units of are hundredths of a decibel, where zero is the original volume of the stream.

Positive decibels correspond to amplification and negative decibels correspond to attenuation. The decibel (db) scale corresponds to the logarithmic hearing characteristics of the ear. An attenuation of 20 db makes a buffer sound half as loud; an attenuation of 40 db makes a buffer sound one quarter as loud. Amplification is not currently supported.

The pan control is cumulative with volume control.

IDirectSoundBuffer::Stop

```
HRESULT Stop(LPDIRECTSOUNDBUFFER lpDirectSoundBuffer);
```

Causes the sound buffer to stop playing.

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

DSERR_INVALIDPARAM	An invalid parameter was passed to the returning function.
DSERR_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to stop playing.

For secondary buffers, the **IDirectSoundBuffer::Stop** member function will set the current position of the buffer to the sample after the last sample played. This means that if **IDirectSoundBuffer::Play** 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 member function will stop the buffer and reset the current position to zero (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 any other cooperative level, this member function simply reverses the effects of **IDirectSoundBuffer::Play**. That is, it configures the primary buffer to stop if no other secondary buffers are playing. If other buffers are playing (in this or other applications), then the buffer will not actually stop until they are stopped. This member function is useful because playing the primary buffer consumes CPU overhead even if the buffer is playing sound data with the amplitude of 0.

IDirectSoundBuffer::Unlock

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

Releases a locked sound buffer.

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

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_PRIOLEVELNEEDED	The caller does not have the priority level required for the function to succeed.

lpDirectSoundBuffer

Address of the DirectSoundBuffer object to unlock.

lpvAudioPtr1

Address of the value retrieved in the *lplpvAudioPtr1* parameter of the **IDirectSoundBuffer::Lock** function.

dwAudioBytes1

Number of bytes actually written into the *lpvAudioPtr1* parameter. It should not exceed the number of bytes returned by the **IDirectSoundBuffer::Lock** function.

lpvAudioPtr2

Address of the value retrieved in the *lplpvAudioPtr2* parameter of the **IDirectSoundBuffer::Lock** function.

dwAudioBytes2

Number of bytes actually written into the *lpvAudioPtr2* parameter. It should not exceed the number of bytes returned by the **IDirectSoundBuffer::Lock** function.

An application must pass both pointers, *lpvAudioPtr1* and *lpvAudioPtr2*, returned by the **IDirectSoundBuffer::Lock** function to ensure the correct pairing of **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Unlock**. The second pointer is needed even if zero bytes were written to the second pointer (that is, if *dwAudioBytes2* equals 0).

Applications must pass the number of bytes actually written to the two pointers in the parameters *dwAudioBytes1* and *dwAudioBytes2*.

The sound buffer must not be locked for long periods.

Structures

 DSBCAPS
 DSBUFFERDESC
 DSCAPS

DSBCAPS

```
typedef struct _DSBCAPS {
    DWORD  dwSize;                // size, in bytes, of this structure
    DWORD  dwFlags;               // see below
    DWORD  dwBufferBytes;         // size, in bytes, of the buffer
    DWORD  dwUnlockTransferRate;  // see below
    DWORD  dwPlayCpuOverhead;     // see below
} DSBCAPS, *LPDSBCAPS;
```

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

dwFlags

Flags specifying buffer-object capabilities. This member can be one or more of the following values:

DSBCAPS_CTRLFREQUENCY	Buffer must have frequency control capability.
DSBCAPS_CTRLPAN	Buffer must have pan control capability.
DSBCAPS_CTRLVOLUME	Buffer must have volume control capability.
DSBCAPS_LOCHARDWARE	Forces the buffer to use hardware mixing, even if DSBCAPS_STATIC is not specified. If this device does not support hardware mixing or the required hardware memory is not available, the call to **IDirectSound::CreateSoundBuffer** will fail. Note that there is no guarantee that a mixing channel will be available for this buffer — the application must make sure of this.
DSBCAPS_LOCSOFTWARE	Forces the buffer to be stored in software (main system) memory and use software mixing, even if DSBCAPS_STATIC is specified and hardware resources are available.
DSBCAPS_PRIMARYBUFFER	Buffer is a primary sound buffer. If not specified, a secondary buffer will be created.
DSBCAPS_STATIC	Indicates that the buffer will be used for static sound data. Typically used for buffers which are loaded once and played many times. These buffers are candidates for hardware (onboard) memory.

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 as no processing is required. For hardware buffers, the buffer might have to be downloaded to the card which may have a limited transfer rate.

dwPlayCpuOverhead

Specifies the CPU overhead as a percentage of main CPU cycles needed to mix this sound buffer. For hardware buffers, this member will be zero 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 similar information to the **DSBUFFERDESC** structure passed to **IDirectSound::CreateSoundBuffer**, with some additional information such as 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 CPU overhead to play the buffer, if 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 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 are used to force the buffer to be located in either hardware or software.

DSBUFFERDESC

```
typedef struct _DSBUFFERDESC
{
    DWORD           dwSize;         // size, in bytes, of this structure
    DWORD           dwFlags;        // see below
    DWORD           dwBufferBytes;  // see below
    DWORD           dwReserved;     // reserved. must be zero
    LPWAVEFORMATEX  lpwfxFormat;    // see below
} DSBUFFERDESC, *LPDSBUFFERDESC;
```

Describes the needed characteristics of a new DirectSoundBuffer object. This structure is used by the **IDirectSound::CreateSoundBuffer** member function.

dwFlags

Flags identifying capabilities to include when creating the new DirectSoundBuffer object. Specify one or more of the following values:

DSBCAPS_CTRLALL	Buffer must have all control capabilities.
DSBCAPS_CTRLDEFAULT	Buffer should have default control options. This is the same as specifying the DSBCAPS_CTRLPAN, DSBCAPS_CTRLVOLUME, and DSBCAPS_CTRLFREQUENCY flags.
DSBCAPS_CTRLFREQUENCY	Buffer must have frequency control capability.
DSBCAPS_CTRLPAN	Buffer must have pan control capability.
DSBCAPS_CTRLVOLUME	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. Note that 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 main system memory (not on the sound card) and use software mixing, even if DSBCAPS_STATIC is specified and hardware resources are available.
DSBCAPS_PRIMARYBUFFER	Buffer is a primary sound buffer.
DSBCAPS_STATIC	Indicates that the buffer will be used for static sound data. Typically used for buffers that are loaded once and played many times. These buffers are candidates for hardware (onboard) memory.

dwBufferBytes

Size, in bytes, of new buffer. Must be zero when creating primary buffers.

lpwfxFormat

Address of a structure specifying the waveform format for the buffer. Must be NULL for primary buffers. You 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, if possible, the buffer to reside in memory

located in the sound card. DSBCAPS_LOCSOFTWARE forces the buffer to reside in main system memory.

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 zero; 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**.

DSCAPS

```
typedef struct _DSCAPS {
    DWORD       dwSize;                         // see below
    DWORD       dwFlags;                        // see below
    DWORD       dwMinSecondarySampleRate;       // see below
    DWORD       dwMaxSecondarySampleRate;       // see below
    DWORD       dwPrimaryBuffers;               // see below
    DWORD       dwMaxHwMixingAllBuffers;        // see below
    DWORD       dwMaxHwMixingStaticBuffers;     // see below
    DWORD       dwMaxHwMixingStreamingBuffers;  // see below
    DWORD       dwFreeHwMixingAllBuffers;       // see below
    DWORD       dwFreeHwMixingStaticBuffers;    // see below
    DWORD       dwFreeHwMixingStreamingBuffers; // see below
    DWORD       dwMaxHw3DAllBuffers;            // see below
    DWORD       dwMaxHw3DStaticBuffers;         // see below
    DWORD       dwMaxHw3DStreamingBuffers;      // see below
    DWORD       dwFreeHw3DAllBuffers;           // see below
    DWORD       dwFreeHw3DStaticBuffers;        // see below
    DWORD       dwFreeHw3DStreamingBuffers;     // see below
    DWORD       dwTotalHwMemBytes;              // see below
    DWORD       dwFreeHwMemBytes;               // see below
    DWORD       dwMaxContigFreeHwMemBytes;      // see below
    DWORD       dwUnlockTransferRateHwBuffers;  // see below
    DWORD       dwPlayCpuOverheadSwBuffers;     // see below
    DWORD       dwReserved1;                     // reserved. do not use
    DWORD         dwReserved2;                      // reserved. do not use
} DSCAPS, *LPDSCAPS;
```

Specifies the capabilities of a DirectSound device, for use by the **IDirectSound::GetCaps** member function.

dwSize

The size, in bytes, of the structure.

dwFlags

Flags specifying device capabilities. This member can be one or more of the following values:

DSCAPS_CONTINUOUSRATE	Device supports all sample rates between the **dwMinSecondarySampleRate** and **dwMaxSecondarySampleRate** values. Typically this means that the actual output rate will be within +/- 10Hz of the requested frequency.
DSCAPS_EMULDRIVER	Device does not have a DirectSound driver installed, so it is being accessed through emulation (that is, through the waveform functions). Applications should expect performance degradation.
DSCAPS_CERTIFIED	This driver has been tested and certified by Microsoft.
DSCAPS_PRIMARY16BIT	Device supports primary buffers with 16-bit samples.
DSCAPS_PRIMARY8BIT	Device supports primary buffers with 8-bit samples.
DSCAPS_PRIMARYMONO	Device supports monophonic primary buffers.
DSCAPS_PRIMARYSTEREO	Device supports stereo primary buffers.
DSCAPS_SECONDARY16BIT	Device supports hardware-mixed secondary

	buffers with 16-bit samples.
DSCAPS_SECONDARY8BIT	Device supports hardware-mixed secondary buffers with 8-bit samples.
DSCAPS_SECONDARYMONO	Device supports hardware-mixed monophonic secondary buffers.
DSCAPS_SECONDARYSTEREO	Device supports hardware-mixed stereo secondary buffers.

dwMinSecondarySampleRate and **dwMaxSecondarySampleRate**

Minimum and maximum sample rate specification that is supported by this device's hardware secondary sound buffers.

dwPrimaryBuffers

Number of primary buffers supported. Will always be 1 for this release.

dwMaxHwMixingAllBuffers, dwMaxHwMixingStaticBuffers, and **dwMaxHwMixingStreamingBuffers**

Description of the hardware mixing capabilities of the device. **dwMaxHwMixingAllBuffers** specifies the total number of buffers that can be mixed in hardware. **dwMaxHwMixingStaticBuffers** specifies the maximum number of static buffers (that is, the buffers located in onboard sound memory), and **dwMaxHwMixingStreamingBuffers** specifies the maximum number of streaming buffers. Note that the first member may be less than the sum of the others, as there are usually some resource trade-offs.

dwFreeHwMixingAllBuffers, dwFreeHwMixingStaticBuffers, and **dwFreeHwMixingStreamingBuffers**

Description of the free (unallocated) hardware mixing capabilities of the device. An application can use these values to determine whether hardware resources are available for allocation to a secondary sound buffer. Also, by comparing these values to the members that specify maximum mixing capabilities, an application can determine whether some hardware resources are already allocated.

dwMaxHw3DAllBuffers, dwMaxHw3DStaticBuffers, and **dwMaxHw3DStreamingBuffers**

Description of the hardware 3D positional capabilities of the device. These will be all be zero for the first release.

dwFreeHw3DAllBuffers, dwFreeHw3DStaticBuffers, and **dwFreeHw3DStreamingBuffers**

Description of the free (unallocated) hardware 3D positional capabilities of the device. These will be all be zero for the first release.

dwTotalHwMemBytes

Size, in bytes, of the amount of memory on the sound card that can store 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 (that is, buffers located in onboard sound memory). This (and the number of bytes transferred) determines the duration of that **IDirectSoundBuffer::Unlock**.

dwPlayCpuOverheadSwBuffers

Description of the CPU overhead (in percentage of CPU) needed to mix software buffers (that is, the buffer located in main system memory). This varies according to the bus type, and the processor type and clock speed.

Note that the unlock transfer rate for software buffers is zero because the data does not need to be transferred anywhere. Similarly, the play CPU overhead for hardware buffers is zero because the mixing is done by the sound device.

Constants

 Error Values

Error Values

DS_OK	The function succeeded.
DSERR_ALLOCATED	The function failed because resources (such as a priority level) were already in use by another caller.
DSERR_ALREADYINITIALIZED	This 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_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	This object does not support aggregation.
DSERR_NODRIVER	No sound driver is available for use.
DSERR_OUTOFMEMORY	The DirectSound subsystem couldn't 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.
E_NOINTERFACE	The requested COM interface is not available.