DirectX Audio

This section provides information about using Microsoft® DirectX® to play and capture sounds in applications.

[C++]
Like other components of DirectX, DirectX Audio can be used with C, C++, and Microsoft® Visual Basic®. This Help file is specific to C and C++. The sample code uses C++ syntax.

[Visual Basic]
Like other components of DirectX, DirectX Audio can be used with C, C++, and Microsoft® Visual Basic®. This Help file is specific to Visual Basic.

Note

The C++ and Visual Basic versions of DirectX Help are based on common source material and share a common table of contents. As a result, some topics in the table of contents contain no information relevant to Visual Basic.

For an overview of the organization of the DirectX Audio Help, see Roadmap.

Roadmap

Information on DirectX Audio is presented in the following sections:

- What's New in DirectX Audio. New features and functionality of this component in DirectX 8.0. If you've used Microsoft® DirectSound® or Microsoft® DirectMusic® before, read this section first, because much has changed since DirectX 7.0.
- Introduction to DirectX Audio. An overview of what DirectX Audio is and what it can do for your application, together with a first look at some important objects and the steps involved in playing audio.
- Understanding DirectX Audio. A deeper look at the underlying mechanisms. This section won't teach you how to implement audio playback or capture in your application, but it will help you understand the application programming interface (API).
- Using DirectX Audio. A guide to using the API. You'll probably want to familiarize yourself with the table of contents for this section, and then refer to parts of it as you need specific information. Use it in conjunction with the reference section.

- Advanced Topics in DirectX Audio. Information of interest mostly for developing specialized applications or applications that need highly optimized performance.
- Programming Tips and Tools. Miscellaneous information about compiling and debugging DirectX Audio applications.

[C++]

- DirectX Audio C++ Tutorials. Step-by-step implementation of basic functionality. If you're the kind of person who learns best by doing, this is a good place to start.
- DirectX Audio C++ Samples. A guide to the sample applications in the SDK, to point you to the sample code you need. As well as showing how to implement basic functionality, each sample demonstrates one or more particular features of DirectX Audio.
- DirectMusic C/C++ Reference. Detailed information about all the API elements declared in the DirectMusic header files. This section includes information about the file format used by DirectMusic and DirectMusic Producer.
- DirectSound C/C++ Reference. Detailed information about all the API elements declared in Dsound.h.

[Visual Basic]

- DirectX Audio Visual Basic Tutorials. Step-by-step implementation of basic functionality. If you're the kind of person who learns best by doing, this is a good place to start.
- DirectX Audio Visual Basic Samples. A guide to the sample applications in the SDK, to point you to the sample code you need. As well as showing how to implement basic functionality, each sample demonstrates one or more particular features of DirectX Audio.
- DirectMusic Visual Basic Reference. Detailed information on the DirectMusic API elements in the DirectX for Visual Basic type library.
- DirectSound Visual Basic Reference. Detailed information on the DirectSound API elements in the DirectX for Visual Basic type library.

DirectMusic Producer

DirectMusic Producer is a powerful authoring application included with the DirectX SDK. It enables composers to create dynamic musical elements that can be incorporated in an application.

If your focus is more on sound design than on application development, you might want to start with DirectMusic Producer Help, which is installed with that application.

If you're an application developer and just want an overview of the material you'll be working with, see Compositional Music Elements.

[C++]

Audio Scripting

DirectX Audio scripting is used by composers and sound designers to have more direct control over the soundtrack in an application. Documentation for the script API is included in the DirectMusic Producer Help.

For an overview, see Audio Scripts. To learn how to incorporate scripts in an application, see Using Audio Scripts.

What's New in DirectX Audio

The audio components of DirectX have undergone a major revision. DirectMusic and DirectSound are now a more unified API for playing sounds of all kinds, and many new capabilities have been added.

The following list describes some of the new features:

Integration of waves and message-based sounds in a single playback mechanism

Wave files and resources can now be loaded by the DirectMusic loader and played by the DirectMusic performance. The application no longer needs to parse the wave and copy or stream it to a DirectSound buffer. The timing of wave playback is based on the DirectMusic master clock, and waves can be synchronized to musical events and processed by tools just like any other segment.

The DirectSound API continues to be supported, and you can still play waves directly through DirectSound buffers. DirectSound continues to be the API for wave capture and full duplex.

More flexible and powerful audiopath model

In previous versions of DirectMusic, performance channels were mapped to ports, and each port sent its output to a single DirectSound buffer. Under the new model, channels in a segment are mapped to audiopaths that control the data flow from the performance to the final output. Output from the synthesizer can be sent to multiple playback buffers, each with its own 3-D positioning and effects. Audiopaths can be created dynamically by the application or authored into segments.

[Visual Basic]

DirectX for Visual Basic applications using DirectMusic must employ the audiopath model. Direct access to ports is no longer supported.

Individual control over segment states

Each playing instance of a segment can have its own audiopath, so parameters such as volume, pan, and pitch bend can be modified for each segment state individually.

DLS2 Synthesis

The DirectMusic synthesizer is now based on the Downloadable Sounds (DLS) Level 2 standard, providing higher-quality sound synthesis. New features of the DLS2 synthesizer include six-stage envelopes, layering of voices, release waveforms, and additional low-frequency oscillators (LFOs). Each voice has an optional low-pass resonant filter.

Special effects

DirectX Media Objects can be attached to sound buffers to add effects. Standard effects such as musical reverb, I3DL2 (Interactive 3-D Audio Level 2) environmental reverb, distortion, and echo are provided by DirectX Audio, and third-party effects can be added.

Greater control over cueing

A composer can set points in a segment that specify where the segment can begin playback, so that the segment can maintain its relationship to the time signature. Primary segments can contain arbitrary points where other segments can be cued, allowing greater control than cueing to the nearest measure, beat, or grid. A new segment cue flag, DMUS_SEGF_SEGMENTEND, causes the segment to play when the current primary segment finishes playing.

Notification of impending silence

The performance notification event has a new notification type, DMUS_NOTIFICATION_MUSICALMOSTEND, which indicates that the last primary segment in the queue is about to end. This notification gives the application a chance to schedule a new segment.

Enhanced MIDI controller support

Curve messages now support RPN (registered parameter number) and NRPN (nonregistered parameter number) controller changes.

Infinite repetition of segments

The new DMUS_SEG_REPEAT_INFINITE flag allows a segment to play for an indefinite time.

Simplified downloading and unloading of bands

Bands can be downloaded and unloaded with a simple method call on the segment object.

Dynamic track data generation

New track configuration and play flags allow tracks to be recomposed each time the segment is played or looped. For example, a chordmap track can be configured to establish a new chord progression each time the segment plays.

[C++]

Audio scripting

Music composers and sound designers can gain greater control over the performance by using scripts. The application does not handle the details of playback but instead calls into the script. For example, a game event might

trigger a script function. The script author can easily modify the soundtrack's response to the game event by modifying the function.

Improved run-time control over tracks
Applications can set individual track configurations to disable playback and parameter control. Tracks can be configured to operate in clock time rather than music time. Tracks in self-controlling segments can be configured to override parameter tracks in controlling or primary segments.

Container object
All components of a DirectMusic Producer project can be kept in a single file, making it easier to find and load all objects. Containers can also be embedded in segments, so that everything necessary to play a segment is kept within the segment.

Lyrics track
Time-stamped text is sent to the performance by a segment containing a lyrics track.

Improved cache management
Applications can more easily release unused objects that were loaded by other objects.

Introduction to DirectX Audio

- The Power of DirectX Audio
- Getting Started with DirectX Audio

The Power of DirectX Audio

DirectX Audio does much more than simply play sounds. It provides a complete system for implementing a dynamic soundtrack that takes advantage of hardware acceleration, Downloadable Sounds (DLS), DirectX Media Objects (DMOs), and advanced 3-D positioning effects.

By using the DirectMusic and DirectSound interfaces in your application, you can do the following:

- Load and play sounds from files or resources in MIDI, wave, or DirectMusic Producer run time format.
- Play from multiple sources simultaneously.
- Schedule the timing of musical events with high precision.
- Send tempo changes, patch changes, and other MIDI events programmatically.
- Use Downloadable Sounds. By using the DLS synthesizer, an application can be sure that message-based music sounds the same on all computers. An application

can also play an unlimited variety of instruments and even produce unique sounds for individual notes and velocities.

- Locate sounds in a 3-D environment.
- Easily apply pitch changes, reverb, and other effects to sounds.
- Use more than 16 MIDI channels. DirectX Audio breaks through the 16-channel limitation and makes it possible for any number of voices to be played simultaneously, up to the limits of the synthesizer.
- Play segments on different audiopaths, so that effects or spatialization can be applied individually to each sound.
- Capture MIDI data or stream ("thru") it from one port to another.
- Capture wave sounds from a microphone or other input.

If you use source files from DirectMusic Producer or a similar application, you can do much more:

- Control many more aspects of playback at run time, for example by choosing a different set of musical variations or altering the chord progression.
- Play music that varies subtly each time it repeats.
- Play waves with variations.
- Map performance channels to audiopaths, so that different parts within the same segment can have different effects.
- Compose wholly new pieces of music at run time, not generated algorithmically but based on components supplied by a human composer.
- Dynamically compose transitions between existing pieces of music.
- Cue transitions, motifs, and sound effects to occur at specified rhythmic points in the performance.

[C++]
These capabilities are the ones most often used by mainstream applications. DirectX Audio is designed to be used easily for the basic tasks, but it also allows low-level access to those who need it. It is also extensible. Specialized applications can implement new objects at virtually every stage on the audiopath, such as the following:

- Loaders to parse data in new or proprietary formats.
- Tracks containing any kind of sequenced data.
- Tools to process messages—for example, to intercept notes and apply transpositions, or to display lyrics embedded in a segment file.
- Custom sequencer.
- Custom synthesizer.
- Effects filters.

Also part of DirectX Audio is DirectMusic Producer, an application that enables composers to create DLS collections, chordmaps, styles, and segments—the pieces that let you take full advantage of the power of DirectMusic. DirectMusic Producer also makes it possible to create playable segments that contain multiple time-stamped waves. These waves can be in compressed or uncompressed format and can be either streamed at run time or wholly contained in memory.

As an application developer, you might never use DirectMusic Producer yourself, but it is a good idea to have a broad understanding of what it does so that you can work effectively with your sound design team. For an introduction from the application designer's perspective, see Compositional Music Elements. For more detail, see the DirectMusic Producer documentation.

DirectX Audio delivers full functionality on Microsoft® Windows® 95, Microsoft® Windows® 98, and Microsoft® Windows® 2000. However, support for hardware synthesizers is available only on Windows 2000 and Windows 98 Second Edition.

Getting Started with DirectX Audio

Creating your first DirectX Audio application does not require a lot of setup. Presumably you have already installed the current version of DirectX and have the paths to the header files set up in your development environment. You must also be able to link to the DirectX library files in the \lib folder. The necessary DLLs and the default DLS collection (used by the software synthesizer) were placed in your system directory as part of the SDK installation.

Apart from that, all you need are a sound card, speakers or headphones, and some sounds. Any of the following will do:

- MIDI (.mid) files.
- Wave (.wav) files.
- Segment (.sgt) files from DirectMusic Producer. Sample files are installed along with that application, and others are included with the DirectX sample applications.

For now, you can ignore the other kinds of files you'll find with DirectMusic Producer. Chordmaps, styles, bands, and DLS files are building blocks for playable segments but aren't playable themselves. Segments can also be obtained from container files, but it requires a bit of extra work to extract them.

More information on getting started is included in the following topics:

- Basic Concepts of DirectX Audio
- Basic Steps in Playing Sounds
- Building DirectX Audio Projects

Basic Concepts of DirectX Audio

This section is an introduction to some of the key concepts of DirectX Audio and the objects that implement them.

DirectX is based on the Component Object Model (COM), so all functionality is performed by objects. In addition, source data is encapsulated in objects.

The objects discussed in this section are exposed in every DirectX Audio application, other than those using only DirectSound. Many other objects are at work behind the scenes and can be accessed by more advanced applications, but they are not discussed here.

The following topics are discussed:

- Loader
- Segments and Segment States
- Performance
- Messages
- Performance Channels
- DLS Synthesizer
- Instruments and Downloading
- Audiopaths and Buffers

Loader

[C++]
The loader (**IDirectMusicLoader8**) is an object that gets other objects. It is generally one of the first objects created in a DirectX Audio application. The loader is used to load all audio content, including DirectMusic segment files, DLS collections, MIDI files, and both mono and stereo wave files. It can also load data stored in resources.

Any object that encapsulates data from a file or resource supports the **IDirectMusicObject8** interface. The loader gets this interface and then uses it to initiate the process of streaming the data into an object in your application. Data objects parse themselves through their implementations of **IPersistStream**, and the entire loading process is carried out automatically. All you need to do is pass a description of the object to the loader, along with a request for the desired interface, such as **IDirectMusicSegment8**.

[Visual Basic]
The loader (**DirectMusicLoader8**) is an object that gets other objects. It is usually one of the first objects created in a DirectX Audio application.

Data objects parse themselves, and the entire loading process is carried out automatically. All you need to do is call the appropriate method of **DirectMusicLoader8**.

Segments and Segment States

Segments are objects encapsulating sequenced sound data. The data might be a MIDI sequence, a wave, a collection of information originating in a segment file from DirectMusic Producer, or a piece of music composed at run time from different components. In general, a segment is a piece of music or other sound that is played as a unit.

A segment can be played as a *primary segment* or *secondary segment*. Only one primary segment can be played at a time. Secondary segments are typically short musical motifs or sound effects played over the primary segment.

Segments originating as MIDI or wave files sound the same each time they are played, unless the application performs some special processing on them. A segment authored in DirectMusic Producer, on the other hand, can contain different musical patterns and other information that allows variation each time the segment is played.

[C++]
Segments can combine different kinds of data such as waves, patterns, chord changes, band changes, and tempo changes. Each type of data is encapsulated in a track object. Applications written in C++ can access individual tracks, but most do not need to. Segments can also contain information about the audiopath on which they should be played, including special effects.

[Visual Basic]
Segments can combine different kinds of data such as waves, patterns, chord changes, band changes, and tempo changes. Each type of data is encapsulated in a track object. Applications written in Visual Basic do not have access to track objects. Segments can also contain information about the audiopath on which they should be played, including special effects.

DirectMusic Producer segments can also contain references to other loadable musical components. For example, it is possible to obtain a band object from a segment authored with that band.

Each time a segment is played, a segment state object is created. The application can use this object to get information about the state of playback and the audiopath for that instance of the playing segment.

Performance

The performance object manages the flow of data from the source to the synthesizer. Typically an application has only a single performance.

The performance handles timing, the mapping of data channels to audiopaths, the routing of messages, tool management, notifications, and other important tasks.

Messages

Audio data flows through the performance in the form of messages. Performance messages are similar to MIDI messages but contain more information and a greater variety of information. A message could contain information about a musical note, a wave, or a controller change. It might even contain text for a display of lyrics.

[C++]
Most applications don't deal directly with messages, which are generated by tracks when a segment is playing. However, it is possible for an application to insert messages into the performance or to intercept messages by implementing tools.

[Visual Basic]
Most applications don't deal directly with messages, which are generated by tracks when a segment is playing. However, it is possible for an application to insert messages into the performance.

Messages are also used for notifications. Applications can request that an event be signaled whenever certain points in the performance are reached—for example, on every beat of the music. Information about the event is contained in a performance message.

Performance Channels

Every playing sound consists of one or more parts. A part might be a MIDI channel, a part in a DirectMusic Producer segment, or a wave. Often a part corresponds to a single musical instrument.

A performance channel is the route between a part and an audiopath. Every message that contains information about a part also specifies the part's performance channel, so that it can be routed correctly.

Performance channels are similar to MIDI channels, but whereas traditional MIDI playback is limited to 16 channels, the number of performance channels is virtually unlimited.

DLS Synthesizer

Data that is not already in the form of a wave—a MIDI note, for example—has to be converted to a wave form before it can be played by the sound card. This conversion is done by a synthesizer.

Although DirectX Audio supports basic MIDI synthesis on some operating systems, most applications use a synthesizer that implements the Downloadable Sounds (DLS) Level 2 standard. In the absence of a suitable hardware synthesizer, the Microsoft software synthesizer is used.

The synthesizer produces sounds based on wave samples and is capable of producing highly sophisticated musical timbres as well as any other kind of sound.

Note

The DLS Level 1 synthesizer used with the DirectX 7 interfaces contains reverbation capabilities, which are on by default. The Waves TrueVerb reverberation technology is licensed to Microsoft Corporation as the SimpleVerb implementation.
The DLS Level 2 synthesizer used with the DirectX 8.0 interfaces does not contain built-in reverberation capabilities. Reverberation is instead implemented as a DMO. Waves MaxxVerb is licenced to Microsoft Corporation for this purpose.

Instruments and Downloading

To play an instrument, the synthesizer needs information about how the instrument should sound. This information, consisting of wave samples and articulation data, is stored in DLS collections. Instrument data is made available to the synthesizer by being downloaded.

Typically a range of notes for one instrument is based on one sample that is appropriately pitched for each note. However, DLS Level 2 enables each note to be based on a different sample or combination of samples. Even the velocity of a note can trigger the use of different samples for that instrument.

By default, the Microsoft software synthesizer takes its DLS data from the Roland GM/GS collection. The default collection contains DLS data for the 128 instruments defined by the General MIDI standard. Custom collections can include instruments of any kind. The wave samples for an instrument do not have to be based on a musical instrument but can be any recorded sound such as a sound effect, a fragment of speech, or even a fully formed measure of music.

Most applications do not need to access collections directly, as the necessary data is either contained in the default collection or referenced by a band object associated with a segment. A band is a set of instruments and settings mapped to performance channels. Several techniques are available for ensuring that band instruments are downloaded before use.

Note

The Roland GM/GS Sound Set cannot be modified. See the Copyright Warning for the legal restrictions.

Wave files and resources also have to be downloaded to the synthesizer before they can be played.

Audiopaths and Buffers

Each DirectMusic segment plays on an audiopath that controls the flow of sounds from the performance to the synthesizer, then through DirectSound buffers where effects can be applied, and finally into the primary buffer, where the final output is mixed.

[C++]

Note

The buffers referred to here are used for streaming and processing PCM data after it has left the synthesizer, and support the **IDirectSoundBuffer8** interface. Another kind of buffer, represented by the **IDirectMusicBuffer8** interface, is used for sequencing message data to the synthesizer. Most applications do not need access to the second kind of buffer, which is managed by the DirectMusic performance.

Applications can create standard audiopaths and then play segments on them. For example, an application could create one audiopath for playing MIDI files to a buffer with musical reverb and another for playing wave files to a buffer with 3-D control.

More sophisticated audiopath configurations can be authored into a segment in DirectMusic Producer. For example, a nonstandard configuration might direct parts in a segment through different DirectSound buffers in order to apply different effects to them.

An audiopath can be seen as a chain of objects through which data is streamed. An application can gain access to any of these objects. For example, you might retrieve a buffer object to set 3-D properties of a sound source, or an effect DMO to change the parameters of the effect.

Basic Steps in Playing Sounds

If you understand the fundamentals outlined in Basic Concepts of DirectX Audio, you're ready to implement a rudimentary soundtrack in an application. After learning these few simple steps, you can go on to explore other parts of the API that will give you as much control over the soundtrack as you need.

[C++]

This topic gives an overview of the necessary steps, without getting into the coding details. For sample code, see Tutorial 1: Playing Audio Files.

To produce a sound, an application needs to do the following:

1. Initialize COM

There are no helper functions for creating DirectMusic objects, so you need to call **CoInitializeEx** to initialize COM.

2. Create and initialize the performance

Most applications have a single performance object. You create it by calling **CoCreateInstance** and obtaining the **IDirectMusicPerformance8** interface. Then call **IDirectMusicPerformance8::InitAudio**. This method can set up a default audiopath.

3. Create the loader

Using **CoCreateInstance**, obtain an **IDirectMusicLoader8** interface. You need to do this only once, and you should keep the same loader object for the life of the application.

4. Load a segment

Call **IDirectMusicLoader8::SetSearchDirectory** so the loader can find the data files. Then call **IDirectMusicLoader8::GetObject** to load a segment from a file or resource and obtain its **IDirectMusicSegment8** interface.

5. Download the band

Download DLS data to the synthesizer so that instruments can play. Wave files also must be downloaded. The simplest way to do this is by calling **IDirectMusicSegment8::Download**.

6. Play the segment

Pass the segment pointer to **IDirectMusicPerformance8::PlaySegmentEx**.

[Visual Basic]
This topic gives an overview of the necessary steps, without getting into the coding details. For sample code, see Tutorial 1: Playing Audio Files.

To produce a sound, an application needs to do the following:

1. Create and initialize the performance

Most applications have a single performance object. You create it by calling **DirectX8.DirectMusicPerformanceCreate**. Then call **DirectMusicPerformance8.InitAudio**. This method can set up a default audiopath.

2. Create the loader

Call **DirectX8.DirectMusicLoaderCreate** to obtain a **DirectMusicLoader8** object. You need to do this only once, and you should keep the same loader object for the life of the application.

3. Load a segment

Call **DirectMusicLoader8.SetSearchDirectory** so the loader can find the data files. Then call **DirectMusicLoader8.LoadSegment** or **DirectMusicLoader8.LoadSegmentFromResource** to obtain a **DirectMusicSegment8** object from a file or resource.

4. Download the band

Download DLS data to the synthesizer so that instruments can play. Wave files also must be downloaded. The simplest way to do this is by calling **DirectMusicSegment8.Download**.

5. Play the segment

Pass the segment pointer to **DirectMusicPerformance8.PlaySegmentEx**.

For a more detailed look at this process, see the following topics:

- Loading Audio Data
- Playing Sounds

Note

It is also possible to play wave sounds using only the DirectSound interfaces. However, doing so requires you to parse the data source yourself and handle the streaming of data to buffers. For more information, see Wave Playback in DirectSound.

Building DirectX Audio Projects

[Visual Basic]
To use DirectX for Visual Basic in a project, you must first ensure that the project has access to the type library.

1. In Visual Studio, on the **Project** menu, click **References**. The **References** dialog box appears.
2. Select the check box next to **DirectX 8 for Visual Basic Type Library**, and then click **OK**.

[C++]
Projects need to include the Dmusici.h header file, which contains declarations for the DirectMusic performance layer. Including this file will bring in three other essential headers:

- Dmusicc.h: declarations for the core layer of DirectMusic
- Dmerr.h: DirectMusic return values
- Dsound.h: the DirectSound API

Dmusicf.h contains file structures and defines, and is needed only for applications such as music-authoring tools that work directly with files and don't rely solely on the loaders built into DirectMusic.

Dmksctrl.h contains declarations for the **IKsControl** interface used for port property sets. You do not need this file if you have the Ksproxy.h and Ks.h files.

Dmplugin.h contains declarations for the **IDirectMusicTool8** and **IDirectMusicTrack8** interfaces, which are implemented by add-ons for advanced applications that need specialized message-processing tools and track types. Most applications do not use this part of the DirectMusic API.

If you are using the DirectSound API directly and want to be able to use helper functions such as **DirectSoundCreate8**, be sure to link to Dsound.lib and include the appropriate folder in the library search path.

You must also ensure that your application has access to the GUIDs used by DirectX Audio. For more information, see Compiling DirectX Samples and Other DirectX Applications.

DirectX Audio uses the multithreading capabilities of the Windows 32-bit operating system. Multithreading allows DirectX to generate, process, and synthesize music in the background while your application is accomplishing other tasks. You should develop your project with multithreading in mind. If nothing else, be sure to link with the multithreaded libraries.

Understanding DirectX Audio

This section describes the underlying workings of DirectX Audio without going into the details of the DirectMusic and DirectSound APIs. It is intended to give you a

deeper understanding of concepts than that presented in Basic Concepts of DirectX Audio. For information on how to use the APIs, see Using DirectX Audio.

The following topics are covered:

- DirectSound and DirectMusic
- Overview of Audio Data Flow
- Compositional Music Elements
- How Music Varies During Playback
- Music Values and MIDI Notes
- Downloadable Sounds
- Audio Scripts

DirectSound and DirectMusic

Previous versions of DirectX presented audio features as two discrete components: DirectSound and DirectMusic. DirectSound was for playing and capturing prerecorded digital samples (waves), and DirectMusic was for playing message-based data ranging from simple MIDI files to musical segments authored in DirectMusic Producer. Although DirectMusic has always been capable of playing nonmusical sound effects through the use of DLS, the emphasis in the past was on its ability to play music.

With DirectX 8, the DirectMusic interfaces are the primary mechanism for loading and playing all sounds, whether they originate as files or resources in wave format, MIDI format, DirectMusic Producer format, or indeed any format for which an add-on loader and tools are available.

Applications can still use the DirectSound interfaces for wave playback, and DirectSoundCapture remains the API for wave capture. However, in most applications DirectSound does its work downstream from the DirectMusic synthesizer. It takes the output from the synthesizer, routes it through effects filters, applies 3-D effects, and does the final mixing before streaming the data to the output device.

If you've used DirectSound in the past, you'll find that switching to the DirectMusic interfaces for loading and playing waves offers the following advantages:

- No need to parse the file or resource. The DirectMusic loader does it for you.
- Automatic use of the Windows audio compression manager (ACM) for wave formats other than those supported by DirectSound.
- Automatic streaming of data. No more manual buffer creation and tracking of read and write pointers.
- Better timing control. Not only can you schedule sounds more precisely using the DirectMusic master clock, but you can easily synchronize sound effects with music.

- Easier management of multiple instances of a sound.

Waves incorporated in DirectMusic Producer files have additional advantages. DLS articulation effects, such as envelopes, can be added to waves. Segments can contain wave variations, adding a random element to playback.

These advantages don't mean you have to lose the low-level control offered by DirectSound. The DirectMusic API enables you to access any object in the audiopath. If you want to get an interface for an existing buffer, you can easily do so. You can also take advantage of all the new features of the DirectSound API for adding filters and effects.

Overview of Audio Data Flow

Typically, a DirectX Audio application obtains musical data from one or more of the following sources:

- MIDI files.
- Wave files.
- Segment files authored in DirectMusic Producer or a similar application.
- Component files authored in an application such as DirectMusic Producer and turned into a complete composition by the composer object.

Note
Any of these data sources can be stored in the application as a resource rather than in a separate file.

Data from these sources is encapsulated in segment objects. Each segment object represents data from a single source. At any given moment in a performance, one primary segment and any number of secondary segments can be playing. Source files can be mixed—for example, a secondary segment based on a wave file can be played along with a primary segment based on an authored segment file.

A segment comprises one or more tracks, each containing timed data of a particular kind—for example, notes or tempo changes. Most tracks generate time-stamped messages when the segment is played by the performance. Other kinds of tracks supply data only when queried by the performance.

The performance first dispatches the messages to any application-defined tools. Such tools are grouped in segment toolgraphs that process only messages from particular segments, audiopath toolsgraphs for messages from all segments playing on the path, and a performance toolgraph that accepts messages from all segments. A tool can modify a message and pass it on, delete it, or send a new message.

[Visual Basic]

Note
Add-on tools are not supported by DirectX for Visual Basic.

Finally, the messages are delivered to the output tool, which converts the data to MIDI format before passing it to the synthesizer. Channel-specific MIDI messages are directed to the appropriate channel group on the synthesizer. The synthesizer creates sound waves and streams them to a device called a sink, which manages the distribution of data through buses to DirectSound buffers.

There are three kinds of buffers:

- *Sink-in buffers* are DirectSound secondary buffers into which the sink streams data. These buffers convert the data format to that of the primary buffer, if necessary, and enable the application to control pan, volume, 3-D spatialization, and other properties. They can also pass their data through effects modules to add effects such as reverberation and echo. The resulting waveform is passed either directly to the primary buffer or to one or more mix-in buffers.
- *Mix-in buffers* receive data from other buffers, apply effects, and mix the resulting wave forms. These buffers can be used to apply global effects. An effect achieved by directing data to a mix-in buffer is called a send.
- The *primary buffer* performs the final mixing on all data and passes it to the rendering device.

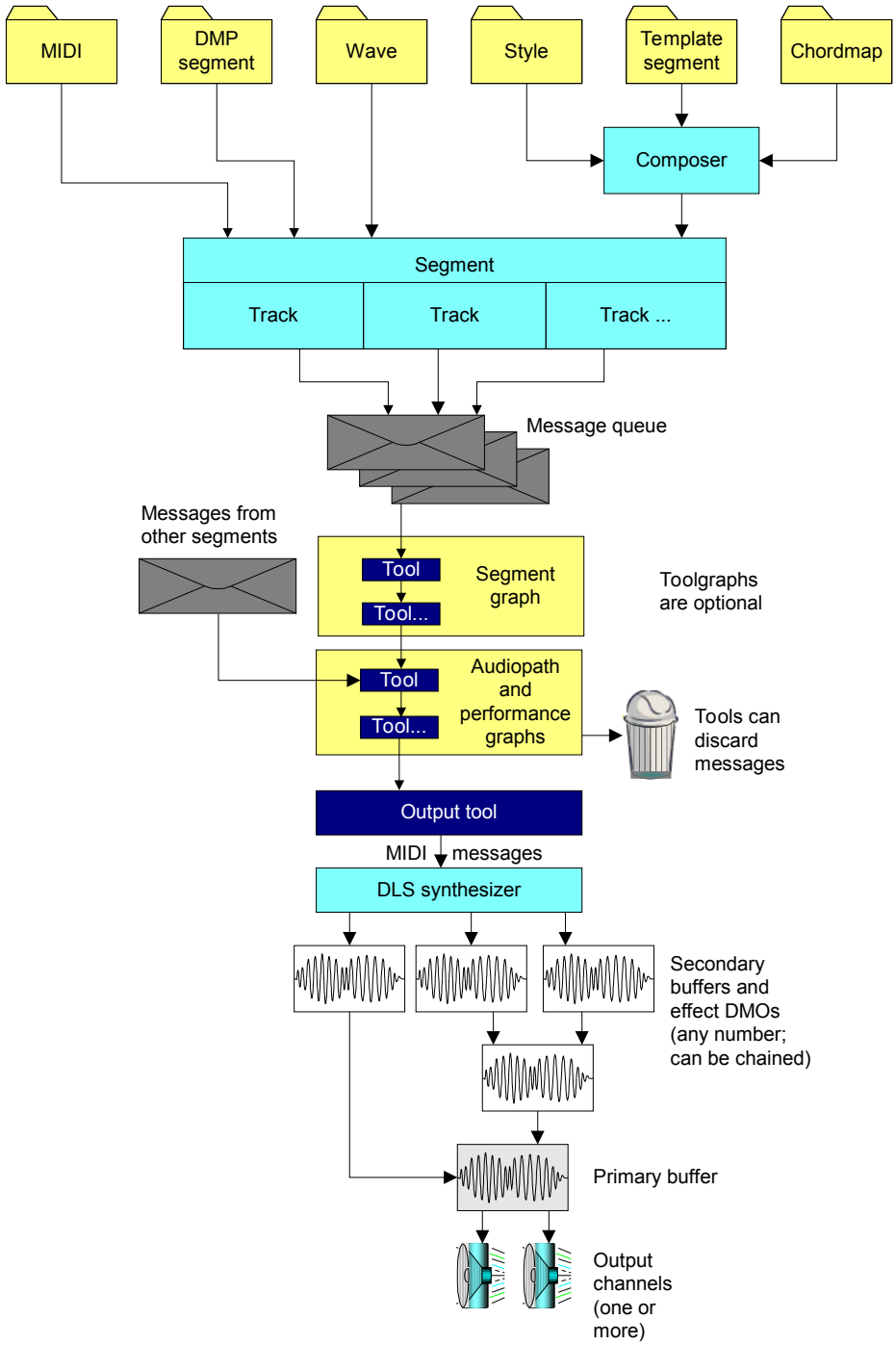
Note

Applications are not responsible for streaming the data through secondary buffers that are part of a DirectMusic performance. Although an application can obtain a buffer object for the purpose of adding effects and changing properties, it cannot lock the buffer and write to it, as it must when using the DirectSound API to play wave sounds.

The following diagram is a simplified view of the flow of data from files to the speakers. A single segment is shown, though multiple segments can play at the same time. The segment gets its data from only one of the four possible sources shown: a wave file, a MIDI file, a segment file authored in DirectMusic Producer, or component files combined by the composer object.

Notes

In all cases, data can come from a resource rather than a file.


MIDI
DMP segment
Wave
Style
Template segment
Chordmap
Composer
Segment
Track
Track
Track ...
Message queue
Messages from other segments
Tool
Tool...
Segment graph
Toolgraphs are optional
Tool
Tool...
Audiopath and performance graphs
Tools can discard messages
Output tool
MIDI messages
DLS synthesizer
Secondary buffers and effect DMOs (any number; can be chained)
Primary buffer
Output channels (one or more)

For a closer look at the flow of messages through the performance, see Using DirectMusic Messages.

For information on how to implement the process shown in the illustration, see Loading Audio Data and Playing Sounds.

Compositional Music Elements

If you want to take full advantage of DirectX Audio, you won't play just MIDI and wave files. You'll take musical elements authored in DirectMusic Producer or a similar application and use them to create performances that can be varied or manipulated in countless ways.

Note

Throughout this documentation, the human composer of musical elements is referred to as the author, to avoid confusion with the composer object of DirectMusic. Similarly, musical elements are said to be authored rather than composed.

The following brief introduction to these elements and how they are authored is meant to give the application developer an understanding of the material being used in the performance. For a more detailed view, see the documentation for DirectMusic Producer. For information on how to incorporate these elements in an application, see Using Compositional Elements.

The following compositional elements are described:

- Styles
- Chordmaps
- Style-based Segments
- MIDI-based Segments
- Templates
- Bands

Styles

A style is a collection of musical patterns that can be used to create a dynamic score. It also contains a time signature and a tempo, and can contain one or more bands.

A pattern is a musical figure, one or more measures long, consisting of a basic sequence of notes for each instrument, or part. These notes are not fixed but are ultimately mapped to particular pitches according to the current key, chord, and play mode. Patterns also include variations.

A *motif* is a special type of pattern designed to be played dynamically over the basic score. Motifs are often used in interactive applications to mark an event.

In DirectMusic Producer, patterns are created by the author on a grid analogous to a piano roll. Each part has its own row, corresponding to a performance channel, on which notes are represented by bars of varying length (duration), thickness (velocity), position on a vertical scale (pitch), and position on a horizontal scale (time).

The author can create many variations for each pattern. Typically, this is done by copying the pattern and then making small changes to one or more parts. At run time, variations are chosen by the style playback mechanism. However, the author can disable any variation for any chord—that is, specify that the variation must never be chosen when a certain chord is being played.

The author also assigns a groove range to the pattern, specifying the groove levels at which the pattern can be played.

The pattern can also be designated as an embellishment. Embellishments are of five types—intro, fill, break, end, and custom—and a pattern can be assigned to one or more of these categories. When the music is played and a certain type of embellishment is called for, only patterns of that type are candidates for playback.

Chordmaps

Much modern music, especially music in the popular, rock, folk, and jazz idioms, is based on the concept of chord progression, meaning that all the notes played within a given span of time are associated with a certain chord, and the music moves harmoniously from one chord to another.

The notes within a pattern authored for DirectMusic are derived from or intended to harmonize with a single chord. At run time, however, the pattern is transposed according to the chord progression—that is, each time the underlying chord changes, DirectMusic modifies the pitch of the notes accordingly.

The chordmap is a road map of chord progressions. Within the chordmap designer, the author chooses chords that can express the desired musical feeling or personality. These chords are arranged in a flowchart along a time line. The time line is conceptually circular, so it keeps looping back to the beginning as long as that segment of music is being played.

Certain important chords are designated as *signposts*. These are chords that must be played at certain points. The music is always moving from one signpost to the next. Between the signposts, however, the chord progression can follow various routes from one chord to another, as mapped out by the author.

The route through the chord chart may be chosen at run time by the composition engine, providing variation in addition to that found in the patterns themselves. Or it may be chosen by the authoring tool when the author is creating a segment.

A chord in the chordmap can actually consist of several different chords, referred to as *subchords*. In order to achieve polytonality by playing different inversions of the same chord, the author can assign different parts to different subchords. Each subchord is valid for one or more levels, and these are matched up with levels assigned to parts in the style.

Style-based Segments

A style-based segment is a largely prebuilt piece of music that the author constructs from the following elements:

- *Style.* A style consists of general information about the music (such as time signature and tempo), as well as patterns. See Styles.
- *Chord progression.* This might be derived automatically by the authoring tool from a chordmap (by choosing a path through the chord chart), or entered manually by the author.
- *Command track*. This track, known as the groove track in DirectMusic Producer, is a series of commands for selecting appropriate patterns at set times. A characteristic of the patterns in styles is that they can be designated as embellishments (intro, fill, break, and end) and can also be assigned a certain groove range by the author. The command track of the segment might instruct the style playback engine to select an intro pattern and play it for the first measure, then play only patterns with a groove level of 25 for the next four measures, then play a break, and so on.
- *Band.* The author can assign instruments and performance channels to all the parts in the various patterns.

MIDI-based Segments

A MIDI-based segment is created in DirectMusic Producer, often by importing a MIDI file to which the author might add tempo, key, and band changes, as well as loop points. Unlike a style-based segment, a MIDI segment has no patterns and no command track. Instead, it has a sequence track that contains MIDI notes and other commands.

Templates

A template is a segment that contains a signpost track and, optionally, a command track. The signpost track can be used to create a new chord progression, either for the segment itself or for another segment that does not have a signpost track. The command track supplies groove levels and embellishments. You can use templates at run time to compose new material.

Note

Templates created in versions of DirectMusic Producer prior to DirectX 8.0 are not playable self-contained segments. They can be used only to compose other segments.

The signpost track contains a sequence of signpost markers, which mark the beginning and end of regions in which variations in the chord progression are possible. Each signpost marker is designated as valid for a particular group of

signpost chords in a chordmap. The author of the content is responsible for assigning signpost markers and signpost chords to the appropriate groups.

DirectMusic composes a segment by applying the signpost track to a particular style and chordmap. Each time the composer encounters a pair of signpost markers along the time line in the template, it searches the chordmap for a pair of signpost chords that belong to that group. If it finds a pair and the interval between them fits into the time available, it follows the chord progression between those two signpost chords, as defined in the chordmap. If it is unable to find a path that works, or if there is no end signpost marker, the engine plays any chord from the group of the beginning signpost marker.

For information on composing segments from templates in an application, see Using Templates.

Bands

A band is a set of instruments, with their performance parameters, associated with particular parts in a piece of music. This isn't the same as a DLS collection, which represents a set of instruments that can be downloaded to the synthesizer and thus made available to any application.

In a tool such as DirectMusic Producer, the author creates a band by assigning instruments to performance channels. These instruments can be from any DLS collection, and instruments from different sources can be mixed within a band. Each of these instruments is given volume, pan, and transposition settings.

Performance channels map instruments to parts. If a pattern calls for a particular note on channel 1, that note is played by the instrument in the current band that is assigned to channel 1. The sound is modified by the band's settings for the volume, pan, and transposition of that instrument.

Bands can be saved as separate files or included in styles or segments.

How Music Varies During Playback

As DirectMusic plays a style-based segment, changes are made to the basic harmony and rhythm so that the performance does not sound static. Changes are partly scripted and partly random.

- *Choice of pattern*. A typical style contains multiple patterns, which are selected in response to commands from the command track. For example, if the command track calls for a break embellishment to be played, the style playback engine selects a break pattern that is compatible with the current groove level. (The author specifies which groove levels are appropriate for each pattern.) If there is more than one suitable pattern, one is chosen according to rules embedded in the segment by the author. The choice might be completely or partly random, or patterns might be selected in a certain sequence.

- *Variations within a pattern*. Any part within a pattern can have multiple variations. Variations can play in an order specified by the author; otherwise the style playback engine makes a random choice of variations on each repetition of the pattern.
- *Groove level*. The groove level of the segment determines which of the patterns in the style can be selected for playback. The current level is set by the command track, which is normally authored into a segment. The groove level of a segment can also be changed programmatically, and a modifier can be applied to all segments by setting the master groove level for the performance.
- *Transposition*. As the segment plays, changes are made to the underlying chord according to the progression in the chord track. The notes in the current pattern are automatically transposed to harmonize with the new chord.
- *Variations in timing*. The playback engine can introduce small random changes in the parameters of individual notes—when they begin and when they end.
- *Band*. The choice of instruments and instrument settings (volume, pan, and transposition) can be changed as the segment is playing, either by the band track within an authored segment or dynamically by the application. The application can change the band by creating a secondary segment based on a band object and then playing that segment, or by setting a parameter on the band track of the playing segment.

In many cases, applications exert control over the music by playing different segments rather than by manipulating existing segments. For example, to have the music reflect a change in the intensity of a game, you can simply transition to a new segment authored for that intensity level. You can achieve a similar effect with a single style-based segment by having the author create patterns with different groove ranges, and then changing the groove level in response to game events.

Music Values and MIDI Notes

Notes in a pattern within a DirectMusic style are not fixed notes. Rather, they are *music values* that become actual notes only when they are transposed to the current chord according to the current *play mode* and *subchord level*.

A music value is a representation of the note's intended role. For example, a music value can specify that a note is intended to be played as the second position in the chord, up one in the scale. When that music value is applied to a particular chord, it is converted to the appropriate MIDI note—the one in the second position in the chord, up one in the scale.

The play mode determines how to interpret the note against the chord. For example, if the mode is DMUS_PLAYMODE_NORMALCHORD, the note is interpreted against the intervals of the chord and scale, based on the root of the chord. If the mode is DMUS_PLAYMODE_FIXEDTOKEY, the note is interpreted as a linear value.

To allow for complex harmonies with multiple parallel chord progressions, DirectMusic chords can comprise multiple subchords. The subchord level is a value in the range from 0 through 31 that determines which subchords of a chord can be used in establishing the music value. Each subchord is valid for one or more levels, as defined by the author of the music. DirectMusic Producer supports up to four subchords per chord.

[Visual Basic]
For an explanation of the data format of music values, see **DMUS_NOTE_PMSG**. For more information on the various play modes, see **CONST_DMUS_PLAYMODE_FLAGS**.

[C++]
When a segment is played, each note is encapsulated in a message structure that specifies the original music value and the final MIDI note along with the play mode and subchord level that were used in transposition. Most applications don't deal directly with note messages, but tools can intercept them and alter the notes. For example, a tool could intercept a note that was transposed in a certain play mode, change the play mode, and calculate a new MIDI value before passing on the message.

More information on music values, play modes, chords, and subchords is contained in the following reference topics:

- **DMUS_NOTE_PMSG**
- **DMUS_PLAYMODE_FLAGS**
- **DMUS_CHORD_PARAM**
- **DMUS_CHORD_KEY**
- **DMUS_SUBCHORD**
- **IDirectMusicPerformance8::MIDIToMusic**
- **IDirectMusicPerformance8::MusicToMIDI**

Downloadable Sounds

In the past, most computer audio has been produced in one of two fundamentally different ways, each with its advantages and disadvantages:

- Waveforms are reproduced from digital samples, typically stored in a .wav file or, in the case of Red Book audio, on a standard CD track. Digital samples can reproduce any sound, and the output is very similar on all sound cards. However, they require large amounts of storage and resources for streaming.

- Musical instrument sounds are synthesized, usually in hardware, in response to messages, typically from a MIDI file. MIDI files are compact and require few streaming resources, but the output is limited to the number of instruments available in the General MIDI set and in the synthesizer, and may sound very different on different systems.

One way to combine the advantages of digital sampling with the compactness and flexibility of MIDI is wave-table synthesis—the synthesis of instrument sounds from digital samples. These samples are obtained from recordings of real instruments and then stored on the hardware. The samples are looped and adjusted in such a way as to produce sounds of any length at various pitches and volumes.

Wave-table synthesis produces more realistic timbres than algorithmic FM synthesis but is still limited to a fixed set of instruments. Moreover, a particular instrument might sound different on different pieces of hardware, depending on the manufacturer's implementation of that instrument.

Enter the Downloadable Sounds (DLS) standard, published by the MIDI Manufacturers Association. DLS is a way of allowing wave-table synthesis to be based on samples provided at run time rather than hard-wired into the system. The data describing an instrument is downloaded to the synthesizer, and then the instrument can be played like any other MIDI instrument. Because DLS data can be distributed as part of an application, developers can be sure that their soundtracks will be delivered uniformly on all systems. Moreover, they are not limited in their choice of instruments.

A DLS instrument is created from one or more digital samples, typically representing single pitches, which are then modified by the synthesizer to create other pitches. Multiple samples are used to make the instrument sound realistic over a wide range of pitches. When a DLS instrument is downloaded, each sample is assigned to a certain range of pitches, called a *region*.

DLS Level 2 allows every note to occupy its own region. Moreover, the timbre for each region can be made up of multiple samples, called layers, and different layers can be triggered depending on the velocity of the note. A single instrument can thus be used to produce thousands of different sounds.

In addition, samples can be given an *articulation*, which defines characteristics that make the sound more like that produced by a real instrument. Articulation includes envelopes for the volume and pitch of the sound and a low-frequency oscillator (LFO) to provide vibrato and tremolo.

Samples can be loopable or single-shot. A loopable sample plays repeatedly for the duration of the note. A single-shot sample plays only once.

DLS data is stored in instrument collections, from which it is downloaded to the synthesizer.

DLS instruments are assigned patch numbers and respond to MIDI messages just like other MIDI instruments. However, a DLS instrument does not have to belong to the General MIDI set. In fact, it doesn't have to represent a musical instrument at all. Any

sound, even a fragment of speech or a fully composed measure of music, can be associated with a DLS instrument.

For more information on DLS collections and how instruments are created, see the documentation for DirectMusic Producer. To learn more about the DLS standard, consult the document "Downloadable Sounds Level 2", available from the MIDI Manufacturers Association.

Most applications don't have to deal directly with instruments or DLS data. The opening of collections and downloading of instrument data is handled by the band object. When you download a band, all the instrument data associated with that band is downloaded as well. For more information, see Using Bands.

Audio Scripts

An audio script is a file containing variables and routines that can be accessed from within an application. It is written in a subset of Microsoft® Visual Basic® Scripting Edition called AudioVBScript. DirectMusic Producer includes an authoring environment for scripts and saves them in a special format that includes binary parameters as well as the text of the script.

Scripts implement some of the key DirectX Audio objects and can perform basic functions such as the following:

- Creating audiopaths
- Setting audiopath volumes
- Setting global parameters for the performance
- Loading, playing, and stopping segments
- Downloading bands

[Visual Basic]
DirectX® for Visual Basic does not enable applications to load and call audio scripts through the API. However, a segment authored in DirectMusic Producer can contain a script track that triggers calls to routines in one or more scripts.

[C++]
DirectX Audio scripts are designed to make it easier for application developers and sound designers to coordinate their efforts. Scripts enable sound designers to have greater and more immediate control over the soundtrack. The basic functionality of loading and playing sounds is performed by the script. The application contains generalized code that calls into the script.

Here's an example of how this might be helpful.

The sound effects for a certain game are stored as individual wave files. The game uses these sounds for events such as weapons firing and monsters grunting. Using

conventional programming techniques, the developers load the individual sounds by file name and play them as secondary segments at appropriate points in the game.

Suppose the sound designers want to make some changes. They decide, for example, that the boss monster should have a different grunt than his minions. They create the necessary files and hand them off to the developers, who implement the changes in code. Considerable time may pass before the sound designers are able to get a newly compiled version of the game and test it.

Now imagine the same scenario using an audio script. Rather than hard-coding the actual sounds into the application, the developers might write code like the following to respond to a grunt. Assume that that *szGrunter* has been set to a string constant such as "Player", "Boss" or "Minion", and that *pdmScript* is an interface to the script object:

```
pdmScript->SetVariableVariant("Grunter", szGrunter, NULL);
pdmScript->CallRoutine("PlayGrunt", NULL);
```

This fragment of code sets the value of the Grunter variable in the script and calls the PlayGrunt script routine. The script author—most likely a member of the sound design team—decides what the routine does. For example, it might test the value of "Grunter" before deciding what sound to play.

To change the response to the game situation, all that is required is an alteration in the text of the script, and the new routine can be tested immediately against the existing application.

The scripting API is documented in the DirectMusic Producer Help file. For more information on how to implement a script in your application, see Using Audio Scripts.

Note

Audio scripts are not designed to be used on Web pages.

Using DirectX Audio

This section is a guide to using the Microsoft® DirectMusic® and Microsoft® DirectSound® APIs in application development.

Information is presented in the following topics:

- Loading Audio Data
- Playing Sounds
- Performance Parameters

- Using Audiopaths
- 3-D Sound
- Using Effects
- Buffer Chains
- Using Compositional Elements
- Using Audio Scripts
- Wave Playback in DirectSound
- Sound Capture

For a more general overview, see Getting Started with DirectX Audio.

For an understanding of the underlying mechanisms of data flow, see Understanding DirectX Audio.

For information on advanced features used mainly by specialized applications, see Advanced Topics in DirectX Audio.

Loading Audio Data

[C++]
Many DirectMusic objects have to be loaded from a file or resource before they can be incorporated into a performance. The **IDirectMusicLoader8** interface is used to manage the enumeration and loading of such objects, as well as to cache them so that they are not loaded more than once.

An application should have only one instance of the loader in existence at a time. You should create a single global loader object and not free it until there is no more loading to be done. This strategy ensures that objects are found and cached efficiently.

The DirectMusic implementation of **IStream** streams the data from the source. The parsing of the data is handled by the various objects themselves through their implementations of **IPersistStream**. As long as you are dealing only with standard DirectMusic data, you don't need to use these interfaces directly.

Loading of objects referenced by other objects is handled transparently. For example, suppose a style being loaded from a DirectMusic Producer file contains a reference to a band whose data is in another file. When the style's **IPersistStream** finds the reference, it obtains the **IDirectMusicGetLoader8** interface from the **IStream** that passed it the data stream. Using this interface, it obtains a pointer to the loader object. Then it calls **IDirectMusicLoader8::GetObject** to load the band.

More information on using the loader is contained in the following topics:

- Setting the Loader's Search Directory
- Scanning a Directory for Objects
- Enumerating Objects

- Loading an Object from a File
- Loading an Object from a Resource
- Containers
- Getting Object Descriptors
- Cache Management
- Garbage Collection
- Setting Objects

See also Custom Loading in the Advanced Topics in DirectX Audio section.

[Visual Basic]
Many DirectMusic objects have to be loaded from a file or resource before they can be incorporated into a performance. The **DirectMusicLoader8** class is used to manage the loading of such objects.

In most cases, an application should have only one **DirectMusicLoader8** object in existence at a time. You should create a single global loader object and not free it until there is no more loading to be done. This strategy ensures that objects are found and loaded efficiently.

However, if your application loads a segment more than once, you should be aware that because of the caching system used internally by DirectMusic, on subsequent loads you might get back the same segment object you were using before, and it will still have any settings you previously gave it, such as start and loop points or a connection to a DLS collection. To ensure that this does not happen, you should release the **DirectMusicLoader8** object by setting it to Nothing, and then create a new one, before reloading the segment.

Loading of objects referenced by other objects is handled transparently. For example, suppose a style being loaded from a DirectMusic Producer file contains a reference to a band whose data is in another file. DirectMusic loads the band automatically, and the application retrieves it by using the **DirectMusicStyle8.GetBand** method.

More information on using the loader is contained in the following topics:

- Setting the Loader's Search Directory
- Loading an Object from a File
- Loading an Object from a Resource
- Containers

Setting the Loader's Search Directory

[C++]

By default, the loader looks for objects in the current directory unless a full path is specified in the **wszFileName** member of the **DMUS_OBJECTDESC** structure describing the object being sought. By using the **IDirectMusicLoader8::SetSearchDirectory** method, you can set a different default path for the **IDirectMusicLoader8::GetObject**, **IDirectMusicLoader8::LoadObjectFromFile**, and **IDirectMusicLoader8::EnumObject** methods. This default path can apply to all objects, or only to objects of a certain class.

The following code example sets the search path for style files:

```
HRESULT mySetLoaderPath (
    IDirectMusicLoader8 *pILoader)    // Previously created.
{
    return pILoader->SetSearchDirectory(
            CLSID_DirectMusicStyle,
            L"c:\\mymusic\\funky",
            FALSE);
}
```

After calling this function, the application can load a style by file name, without including the full path.

[Visual Basic]
By default, the loader looks for file objects in the current directory unless a full path has been passed to the load method. Using the **DirectMusicLoader8.SetSearchDirectory** method, you can set a different default path for the **DirectMusicLoader8.LoadBand**, **DirectMusicLoader8.LoadCollection**, **DirectMusicLoader8.LoadSegment**, and **DirectMusicLoader8.LoadStyle** methods.

Scanning a Directory for Objects

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The **IDirectMusicLoader8::ScanDirectory** method scans the current search directory for objects of a given class. You can further narrow the search by providing a subclass and a file extension other than "*".

The method compiles a list of all matching files and uses the **IDirectMusicObject8::ParseDescriptor** method to extract the GUID and the name

of the object. These identifiers are retained in an internal database so that the application can subsequently load objects by GUID or name rather than by file name. See Loading an Object from a File.

Note

If you are working with DirectMusic Producer content, it is always a good idea to call **IDirectMusicLoader8::ScanDirectory** before loading any objects. Even though you may be loading objects explicitly by file name, those objects might contain references to other objects not identified by file name, and the loader will not be able to find these referenced objects if **ScanDirectory** has not been called on every directory in which the objects might be.

If you include a pointer to a string in the *pwszScanFileName* parameter of the **ScanDirectory** method, the results of the scan are cached in a file by that name to speed up subsequent scans. When a cache file is available, the method updates object information only for files whose time stamps or sizes have changed.

Note

In the current version of DirectMusic, **ScanDirectory** does not use the cache file. Nevertheless, you can implement a cache file now, and it will speed up performance under future versions.

For an example, see Enumerating Objects.

Enumerating Objects

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Use the **IDirectMusicLoader8::EnumObject** method to iterate through all objects of a given class, or of all classes, that have previously been listed in the internal database through a call to **IDirectMusicLoader8::ScanDirectory** or calls to **IDirectMusicLoader8::GetObject**. A description of each object found is returned in a **DMUS_OBJECTDESC** structure.

Note

To be sure of finding all objects, call **ScanDirectory** first. **EnumObject** works by checking the internal database of objects, not by parsing disk files.

The following example enumerates all listed style objects in the current search directory and displays information about each one by using the **TRACE** debugging macro. The loop continues until there are no more objects of that class to enumerate.

```
void myListStyles(
```

```
    IDirectMusicLoader *pILoader)

{
  HRESULT hr = pILoader->SetSearchDirectory(
      CLSID_DirectMusicStyle,
      L"c:\\mymusic\\wassup",
      TRUE);
  if (SUCCEEDED(hr))
  {
    hr = pILoader->ScanDirectory(
        CLSID_DirectMusicStyle,
        L"sty",
        L"stylecache");
    if (hr == S_OK)    // Only if files were found.
    {
      DWORD dwIndex;
      DMUS_OBJECTDESC Desc;
      Desc.dwSize = sizeof(DMUS_OBJECTDESC);
      for (dwIndex = 0; ;dwIndex++)
      {
        if (S_OK ==(pILoader->EnumObject(
            CLSID_DirectMusicStyle,
            dwIndex, &Desc)))
        {
          TRACE("Name: %S, Category: %S, Path: %S\n",
              Desc.wszName,
              Desc.wszCategory,
              Desc.wszFileName);
        }
        else break;
      }
    }
  }
}
```

Notice that the example does not use the **SUCCEEDED** macro to test the result of the method call, because **EnumObject** returns a success code, S_FALSE, for an invalid index number.

Loading an Object from a File

[C++]
To load an object, first obtain the **IDirectMusicLoader8** interface, as in the following example:

```
IDirectMusicLoader8* m_pLoader;

CoInitializeEx(NULL, 0);
HRESULT hr = CoCreateInstance(
    CLSID_DirectMusicLoader,
    NULL,
    CLSCTX_INPROC,
    IID_IDirectMusicLoader8,
    (void**)&m_pLoader);
```

You can then load an object from file by using either **IDirectMusicLoader8::LoadObjectFromFile** or **IDirectMusicLoader8::GetObject**. The first of these methods is more convenient because it does not require you to describe the object by filling out a **DMUS_OBJECTDESC** structure.

The following sample code loads four segments from a directory previously set by **IDirectMusicLoader8::SetSearchDirectory**:

```
IDirectMusicSegment8 * m_pSegments[4];

static WCHAR  wszNames[4][MAX_PATH] = {
    L"AudioPath1.sgt",
    L"AudioPath2.sgt",
    L"AudioPath3.wav",
    L"AudioPath4.sgt"
};

for (DWORD dwIndex = 0; dwIndex < 4; dwIndex++)
{
    hr = m_pLoader->LoadObjectFromFile(
            CLSID_DirectMusicSegment,
            IID_IDirectMusicSegment8,
            wszNames[dwIndex],
            (void**) &m_pSegments[dwIndex]);
}
```

The following example function uses **IDirectMusicLoader8::GetObject** to load a style object from a file. The first parameter receives a pointer to the style.

```
void myLoadStyle(
    IDirectMusicStyle8 **ppStyle, IDirectMusicLoader8 *pLoader)
{
    if (pLoader)
    {
        DMUS_OBJECTDESC Desc;

        // Start by initializing Desc with the file name and
```

```
        // class GUID for the style object.

        wcscpy(Desc.wszFileName,L"c:\\mymusic\\funky\\polka.sty");
        Desc.guidClass = CLSID_DirectMusicStyle;
        Desc.dwSize = sizeof (DMUS_OBJECTDESC);
        Desc.dwValidData = DMUS_OBJ_CLASS |
                    DMUS_OBJ_FILENAME |
                    DMUS_OBJ_FULLPATH;

        pLoader->GetObject(&Desc, IID_IDirectMusicStyle8,
            (void **) ppStyle);
    }
}
```

The example identifies the file by a full path name and indicates this by setting the DMUS_OBJ_FULLPATH flag.

To identify the particular file object being sought, fill in at least one of the **wszName**, **guidObject**, and **wszFileName** members of the **DMUS_OBJECTDESC** structure, and set the corresponding flag or flags in the **dwValidData** member. If you identify the file by **wszName** or **guidObject**, but not by **wszFileName**, you must first call the **IDirectMusicLoader8::ScanDirectory** method to make the GUIDs and names in the current directory available. For more information, see Scanning a Directory for Objects.

[Visual Basic]
To load an object, first create a **DirectMusicLoader8** object. Then call one of the following methods:

- **DirectMusicLoader8.LoadBand**
- **DirectMusicLoader8.LoadChordMap**
- **DirectMusicLoader8.LoadCollection**
- **DirectMusicLoader8.LoadSegment**
- **DirectMusicLoader8.LoadStyle**

Pass in either a simple file name or a full path for the file that contains the desired object.

The following code example, in which *objDX* is a **DirectX8** object, loads a segment from a file in the current directory:

```
Dim objDMLoader As DirectMusicLoader8
Dim objSeg As DirectMusicSegment8

Set objDMLoader = objDX.DirectMusicLoaderCreate
Set objSeg = objDMLoader.LoadSegment("Myseg.sgt")
```

See also Loading an Object from a Resource.

Loading an Object from a Resource

Objects stored as resources or at some other location in memory are loaded in much the same way as file objects. See Loading an Object from a File.

[C++]
With memory objects, however, the **wszName**, **guidObject**, and **wszFileName** members of the **DMUS_OBJECTDESC** structure are irrelevant. Instead, you must obtain a pointer to the block of memory occupied by the object, and its size, and put these in the **pbMemData** and **llMemLength** members respectively. You must also set the DMUS_OBJ_MEMORY flag in the **dwFlags** member.

The memory cannot be released once **IDirectMusicLoader8::GetObject** has been called, because the loader keeps an internal pointer to the memory to facilitate caching data. If you want to clear it out, call **IDirectMusicLoader8::SetObject** with the same **DMUS_OBJECTDESC** descriptor, but with NULL in **pbMemData**. This is not an issue when loading from a resource, because resource memory is not freed.

The following function loads a MIDI file from a resource into a segment:

```
// m_pLoader is a valid IDirectMusicLoader8 pointer.

IDirectMusicSegment8* m_pSegment;

HRESULT LoadMidi(HMODULE hMod, WORD ResourceID)
{
    HRESULT             hr;
    DMUS_OBJECTDESC     ObjDesc;

    HRSRC hFound = FindResource(hMod,
        MAKEINTRESOURCE(ResourceID), RT_RCDATA);
    HGLOBAL hRes = LoadResource(hMod, hFound);

    ObjDesc.dwSize = sizeof(DMUS_OBJECTDESC);
    ObjDesc.guidClass = CLSID_DirectMusicSegment;
    ObjDesc.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_MEMORY;
    ObjDesc.pbMemData = (BYTE *) LockResource(hRes);
    ObjDesc.llMemLength = SizeofResource(hMod, hFound);

    hr = m_pLoader->GetObject(
        &ObjDesc, IID_IDirectMusicSegment8,
        (void**) &m_pSegment );

    return hr;
```

```
}
```

Objects referenced by other objects must be loaded first. For example, if you load a segment that contains a reference to a style, the style must already be loaded in order for the segment to play correctly. Alternatively, you can call **IDirectMusicLoader8::SetObject** on the style so that the segment can find it.

[Visual Basic]
The following methods each take a module name and resource identifier as parameters and return an instance of the appropriate class:

- **DirectMusicLoader8.LoadBandFromResource**
- **DirectMusicLoader8.LoadChordMapFromResource**
- **DirectMusicLoader8.LoadCollectionFromResource**
- **DirectMusicLoader8.LoadSegmentFromResource**
- **DirectMusicLoader8.LoadStyleFromResource**

The following resource types are recognized by the loader:

String Identifier	Method
DMBAND	**LoadBandFromResource**
DMCHORD	**LoadChordmapFromResource**
DMCOLL	**LoadCollectionFromResource**
DMSEG	**LoadSegmentFromResource**
DMSTYLE	**LoadStyleFromResource**

Objects referenced by other objects must be loaded first. For example, if you load a segment that contains a reference to a style, the style must be loaded first in order for the segment to play correctly.

The following code example, where *loader* is a **DirectMusicLoader8** object and *perf* is a **DirectMusicPerformance8** object, loads and plays a MIDI file stored as a DMSEG resource in the executable:

```
Dim seg As DirectMusicSegment8
Set seg = loader.LoadSegmentFromResource("listen.exe", "CANYON.MID")
Call seg.Download(perf)
Call perf.PlaySegmentEx(SEG, 0, 0)
```

Containers

[C++]

Containers are objects representing files that contain various objects. A container file might hold all the data necessary for a performance, including segments, styles, and DLS collections. Container files are typically created in DirectMusic Producer. Containers can also exist within segment and script files.

You load a container like any other object, using **IDirectMusicLoader8::GetObject**. This method makes all objects in the container known to the loader, so that you can then use **GetObject** to retrieve them by name or GUID.

After you have obtained the **IDirectMusicContainer8** interface, you can enumerate the objects in the container by using **IDirectMusicContainer8::EnumObject**.

The following sample function loads a container, retrieves a segment from it by name, and returns an **IDirectMusicSegment** interface. For purposes of demonstration, the container object is created and released within the function; in practice, this should be done only once during the life of the application, to prevent duplication of objects.

```
IDirectMusicSegment* LoadSegmentFromContainer (
      IDirectMusicLoader8* pLoader,
      WCHAR* wszFileName,
      WCHAR* wszSegmentName )

{
  DMUS_OBJECTDESC ObjDesc;
  IDirectMusicSegment* pSegment = NULL;
  IDirectMusicContainer8* pContainer = NULL;

// Load the container.

  HRESULT hr = pLoader->LoadObjectFromFile(CLSID_DirectMusicContainer,
    IID_IDirectMusicContainer8, wszFileName, (void**)&pContainer);
  if (FAILED(hr))
      return NULL;

// Describe the segment.

  ZeroMemory(&objDesc, sizeof(objDesc));
  objDesc.dwSize = sizeof(objDesc);
  objDesc.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_NAME;
  objDesc.guidClass = CLSID_DirectMusicSegment;
  wcsncpy (objDesc.wszName, wszSegmentName,
    sizeof(objDesc.wszName) - 1);
  objDesc.wszName[sizeof(objDesc.wszName) - 1] = 0;

// Load the segment.

  hr = pLoader->GetObject(&ObjDesc, IID_IDirectMusicSegment,
    (void**) &pSegment);
```

```
// Release the container from the cache and destroy the object.

  if (pContainer)
  {
    IDirectMusicObject *pObject = NULL;
    pContainer->QueryInterface(IID_IDirectMusicObject,
          (void **)&pObject);
    if (pObject)
    {
      pLoader->ReleaseObject(pObject);
      pObject->Release();
    }
  pContainer->Release();
  }
  if (S_OK != hr) return NULL;
  return pSegment;
}
```

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not support containers.

Getting Object Descriptors

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Once you have loaded an object, you can use its **IDirectMusicObject8** interface to retrieve information about it in a **DMUS_OBJECTDESC** structure.

The following code example uses the **IDirectMusicObject8::GetDescriptor** method to obtain the name of a style:

```
//  pStyle is a valid pointer to an IDirectMusicStyle8 interface.

IDirectMusicObject8 *pIObject;
DMUS_OBJECTDESC Desc;

if (SUCCEEDED(pStyle->QueryInterface(IID_IDirectMusicObject8,
      (void **) &pIObject)))
```

```
{
    if (SUCCEEDED(pIObject->GetDescriptor(&Desc)))
    {
        if (Desc.dwValidData & DMUS_OBJ_NAME)
        {
            // Desc.wszName contains the name of the style.
        }
    }
    pIObject->Release();
}
```

Cache Management

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
When an object is cached, the same instance of the object is always returned by the **IDirectMusicLoader8::GetObject** method.

The cache stores a pointer to the object. The memory for the object itself is managed by COM, and is not released until the reference count reaches zero. It is important to remember that releasing an object from the cache is not the same as releasing your application's COM reference to it.

Caching is used extensively in the file-loading process to resolve links to objects. For example, two segments could reference the same style. When the first segment loads, it calls the loader to get the style. The loader creates a style object, loads the data from disk, caches a pointer to the style object, and returns this pointer to the segment. If caching is enabled, when the second segment loads, it asks for the style, and the loader immediately returns the same pointer. Now both segments point to the same style. If caching is disabled, the second segment's request for the style causes a duplicate style object to be loaded from the file, at a cost in time and memory.

Here's another example. A band object counts on the loader to keep the General MIDI DLS collection cached. Every time a band has to download a GM instrument, it gets the collection from the loader. If caching for CLSID_DirectMusicCollection is disabled, every patch change in a MIDI file causes a separate copy of the entire GM collection to be created. This is obviously undesirable.

By default, caching is enabled for all object classes. You can disable caching for an object class, or for all objects, by using the **IDirectMusicLoader8::EnableCache** method. This method can also be used to re-enable caching for any or all object classes.

If you want to clear the cache without disabling future caching, use the **IDirectMusicLoader8::ClearCache** method. It's not necessary to call this method before terminating your application, because the cache is automatically cleared when the loader is released. **ClearCache** is only useful if the application soundtrack is changing completely, with all new instruments and source files.

To cache a single object when general caching is disabled, pass it to the **IDirectMusicLoader8::CacheObject** method.

You can remove an object from the cache, ensuring that it will be loaded again on the next call to **GetObject**, by using the **IDirectMusicLoader8::ReleaseObject** or **IDirectMusicLoader8::ReleaseObjectByUnknown** method. It is a good idea to call one of these methods before calling **Release** on an object, especially a segment. If you don't, a reference to the object remains in the cache, so the object continues to exist. As well as taking up memory, the object might retain certain state information. In the case of a segment, any instance that you load later will be taken from the cache, and the start point and loop points will be the same as they were when the previous instance was destroyed.

For more information on cache management, see Garbage Collection.

Garbage Collection

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Caching of loaded objects can lead to wasted memory when an application loads many objects, particularly objects that reference other objects.

When automatic caching is enabled, as it is by default, every object loaded by **IDirectMusicLoader8::GetObject** is cached, including objects that are loaded by reference. For example, if you call **GetObject** on a segment, and that segment contains a reference to a script, the script is loaded and cached as well.

When you call **IDirectMusicLoader8::ReleaseObject** or **IDirectMusicLoader8::ReleaseObjectByUnknown**, however, only the primary object that was loaded by **GetObject** is removed from the cache. Referenced objects are not released, regardless of whether they are still in use by other objects.

In order to clean up objects that are not in use, call **IDirectMusicLoader8::CollectGarbage**. This method releases all objects from the cache except objects directly loaded by **GetObject** and objects referenced by them. **CollectGarbage** clears an object from the cache by releasing the loader's COM reference to the object. If the object's reference count drops to zero as a result, the object destroys itself, thus making its memory available again.

In summary, to ensure that loaded objects do not remain in memory when no longer needed, you need to do the following:

- Call **ReleaseObject** or **ReleaseObjectByUnknown** on any object for which **GetObject** has been called.
- Call **CollectGarbage** to release the loader's reference to any objects that were loaded indirectly.
- Call **Release** on any pointers held by your application.

A complication arises when objects have circular references to one another. Suppose a segment contains a reference, by way of its script track, to a script object, and the script contains a reference to the segment. You load the segment directly by calling **GetObject**, and the script is loaded indirectly. Then you release the segment from the cache by using **ReleaseObject,** and call **Release** on your application's reference to it. The segment continues to exist because there is still one COM reference to it, which is held by the script object. The script is now garbage, because it is not referenced by any other object in the cache. Without taking special measures, however, **CollectGarbage** could only release the loader's reference to the script; therefore its reference count would not drop to zero. The segment and script would continue to be referenced by one another, and although both were removed from the cache, they would both continue to exist in memory.

To avoid this problem, **CollectGarbage** calls an internal method on an object that forces the object to release its references to other objects. In the example above, it causes the script to release its reference to the segment. The segment's reference count drops to zero, and in the course of destroying itself, the segment releases its reference to the script, thus allowing the script to destroy itself when the loader releases its reference.

There is one more complication, however. Suppose the application has obtained an interface to the script that the loader knows nothing about, and neglects to call **Release** on this pointer. The script continues to exist, but it might not be able to behave as it should, because it no longer has a reference to the segment. Calling a method on the script could lead to a fatal error. To prevent this, **CollectGarbage** ensures that all methods on the script return DMUS_S_GARBAGE_COLLECTED.

This scenario does not affect most applications. However, you should be aware that calling a method on an object that has been cleared from the cache by **CollectGarbage** might not yield the desired result.

The following sample code, where *m_pLoader* is an **IDirectMusicLoader8** interface and m_pPerformance is an **IDirectMusicPerformance8** interface, loads a script that contains a reference to a segment. After calling a routine in the script, the example removes the script object from the cache and then calls **CollectGarbage** to free the segment object. If the segment contains a reference to the script, this is released so that the script can be destroyed, in turn releasing the segment and allowing it to be destroyed.

```
// Load script and call routine.
```

```
IDirectMusicScript8 *pScript;

m_pLoader->GetObject(scriptdesc, IID_IDirectMusicScript8, &pScript);
pScript->Init(m_pPerformance, NULL);
pScript->CallRoutine(L"DoorSlam", NULL);

// Release script object and collect garbage.

pLoader->ReleaseObjectByUnknown(pScript);
pLoader->CollectGarbage();
pScript->Release();
```

Setting Objects

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Sometimes it is desirable to tell the loader where to get an object, without actually loading that object, so that the loader can retrieve it if the object is later referenced by other objects as they are being loaded. You might also want to give an object a new attribute so that the loader can find it by that attribute.

The **IDirectMusicLoader8::SetObject** method takes as a parameter a **DMUS_OBJECTDESC** structure that contains two key pieces of information:

- A pointer to the data. This can be either a file path or a pointer to a block of memory. See Loading an Object from a File and Loading an Object from a Resource.
- An identifier for the object when it is referenced later. This could be a GUID or a name. Later, the call to **IDirectMusicLoader8::GetObject** will find the stored object by using the same name or GUID. Note that you cannot change a GUID or name that already exists in the object.

On return, the **DMUS_OBJECTDESC** structure may contain additional information about the object gathered by the loader.

The following function assigns a name to an unnamed object (such as a MIDI file) in a resource:

```
// m_pLoader is a valid IDirectMusicLoader8 interface pointer.

HRESULT SetObjectFromResource(const GUID* guid, int ID,
    char* type, WCHAR* name)
```

```
{
  HRSRC hResource = NULL;
  HGLOBAL hData = NULL;
  hResource = FindResource(g_hInstance, MAKEINTRESOURCE(ID), type);
  if (hResource != NULL)
  {
    hData = LoadResource(g_hInstance, hResource);
    if (hData != NULL)
    {
      DMUS_OBJECTDESC desc;
      if(m_pLoader && (hResource != NULL) && (hData != NULL))
      {
        ZeroMemory(&desc,sizeof(desc));
        desc.pbMemData = (BYTE*) LockResource(*hData);
        desc.llMemLength = SizeofResource(g_hInstance, (*hResource));
        desc.guidClass = (*guid);
        desc.dwSize = sizeof(desc);
        desc.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_MEMORY;
        if (name)
        {
           wcscpy(desc.wszName, name);
           desc.dwValidData |= DMUS_OBJ_NAME;
        }
        return m_pLoader->SetObject(&desc);
      }
    }
  }
  return E_FAIL;
}
```

The sample function could be used to assign a name to a MIDI file stored as a resource of type "MIDI", as in the following function call:

```
SetObjectFromResource(CLSID_DirectMusicSegment, 101,
      "MIDI", "canyon");
```

The object can now be loaded at any time by name.

Playing Sounds

This section introduces the basic elements of a DirectMusic performance and the key methods that you need to get data from source to output.

The following topics are discussed:

- Creating the Performance
- Using Segments
- Using Bands
- Timing
- Notification and Event Handling

Creating the Performance

[C++]
The performance object is the overall manager of playback. Among the tasks it performs are the following:

- Managing ports and audiopaths
- Mapping channels to audiopaths
- Playing and stopping segments
- Dispatching messages
- Managing tools and timing

Most applications have a single performance object, but it is possible to have more than one performance with different parameters, such as master tempo or volume.

The following code example creates a performance and obtains a pointer to the **IDirectMusicPerformance8** interface:

```
IDirectMusicPerformance8* pPerf;

if (FAILED(CoCreateInstance(
      CLSID_DirectMusicPerformance,
      NULL,
      CLSCTX_INPROC,
      IID_IDirectMusicPerformance8,
      (void**)&pPerf
   )))
{
   pPerf = NULL;
}
```

After the performance is created, it must be initialized. If your application is using audiopaths, as is recommended, you must call the **IDirectMusicPerformance8::InitAudio** method. Applications using the earlier channel-to-port mapping model call **IDirectMusicPerformance8::Init** instead. For more information, see Migrating from Ports to Audiopaths.

An important part of initialization is the creation of a DirectMusic object. You can pass an existing **IDirectMusic8** interface pointer to **IDirectMusicPerformance8::InitAudio**, but in most cases it is more convenient to

have **InitAudio** create the DirectMusic object. You can also choose whether or not to retrieve a pointer to the **IDirectMusic8** interface, depending on how much control you need over ports and the master clock. Most applications don't need access to the methods of **IDirectMusic8** and can pass NULL as the *ppDirectMusic* parameter of **InitAudio**.

InitAudio can also take an existing DirectSound object. DirectSound manages the sound data after it leaves the synthesizer. In most cases you can let **InitAudio** create this object. You don't need an interface to it unless you intend to use DirectSound for other purposes such as creating a DirectSoundCapture object or for playing waves directly into buffers rather than through the DirectMusic performance.

By passing a **DMUS_AUDIOPARAMS** structure to **InitAudio**, the application can request synthesizer capabilities or set a synthesizer other than the default one. Most applications don't need to do this.

The following code example initializes the performance without retrieving pointers to the DirectMusic and DirectSound objects. It creates a standard default audiopath with 16 performance channels and all available features on the port.

```
// hWnd is the application window handle

if (SUCCEEDED(pPerf->InitAudio(NULL, NULL, hWnd,
      DMUS_APATH_SHARED_STEREOPLUSREVERB, 16,
      DMUS_AUDIOF_ALL, NULL)))
{
   // Performance initialized.
}
```

[Visual Basic]
The performance object is the overall manager of playback. Among the tasks it performs are the following:

- Managing ports and audiopaths
- Mapping channels to audiopaths
- Playing and stopping segments
- Dispatching messages
- Managing the timing

Most applications have a single **DirectMusicPerformance8** object, but it is possible to have more than one performance with different parameters, such as master tempo or volume.

After the performance is created, it must be initialized by a call to **DirectMusicPerformance8.InitAudio**.

InitAudio associates a **DirectSound8** object with the performance so that DirectSound can manage the sound data after it leaves the synthesizer. If your application uses DirectSound independently, you can either pass an existing object to **InitAudio** or set a variable to the object created by the method. In most cases, however, you don't need access to the **DirectSound8** object.

By setting values in the **DMUS_AUDIOPARAMS** type that must be passed to **InitAudio**, the application can request synthesizer capabilities or set a synthesizer other than the default one. Most applications don't need to do this, and can request the default synthesizer and capabilities by leaving **DMUS_AUDIOPARAMS.lValidData** at 0.

The follow code example, in which *objDX* is a **DirectX8** object, creates a performance and initializes it. It creates a standard default audiopath with 16 performance channels and all available features on the port.

```
Dim objDMPerformance as DirectMusicPerformance8
Dim audParams As DMUS_AUDIOPARAMS

Set objDMPerformance = objDX.DirectMusicPerformanceCreate
objDMPerformance.InitAudio Me.hWnd, DMUS_AUDIOF_ALL, audParams, _
    Nothing, DMUS_APATH_SHARED_STEREOPLUSREVERB, 16
```

Using Segments

[C++]
Segments are the basic units of playable data in the DirectMusic performance. A segment is represented by an **IDirectMusicSegment8** interface.

You can create a segment object in an application by any of the following means:

- Load a file or resource object that supports the **IDirectMusicSegment8** interface. For more information, see Loading Audio Data.
- Get a motif from a style by using the **IDirectMusicStyle8::GetMotif** method.
- Use methods of the **IDirectMusicComposer8** interface to create a composition or transition at run time. See Overview of Programming for Composition and Using Transitions.
- Make a copy of an existing segment by using the **IDirectMusicSegment8::Clone** method.
- Use the **IDirectMusicBand8::CreateSegment** method. This creates a special type of secondary segment that is used only for making band changes. See Making Band Changes Programmatically.
- Use the **IDirectMusicPatternTrack8::CreateSegment** method to create a segment from a pattern track object. Most applications don't do this, because pattern track objects usually come from segments in the first place.

- Construct a segment from existing tracks. Create a segment object by calling **CoCreateInstance**, and then add tracks by calling **IDirectMusicSegment8::InsertTrack**. This technique is not used by most applications.

Each segment consists of one or more tracks, each represented by an **IDirectMusicTrack8** interface. Tracks contain most of the data for the segment, whether that data consists of note events, band changes, tempo changes, or other timed events. Applications generally don't need to use this interface, because the tracks are managed through the segment object. For more information, see DirectMusic Tracks.

[Visual Basic]
Segments are the basic units of playable data in the DirectMusic performance. A segment is represented by a **DirectMusicSegment8** object.

You can create a segment object in any of the following ways:

- Load a file or resource object that supports the **DirectMusicSegment8** class by using the **DirectMusicLoader8.LoadSegment** or the **DirectMusicLoader8.LoadSegmentFromResource** method. For more information, see Loading Audio Data.
- Get a motif from a style by using the **DirectMusicStyle8.GetMotif** method.
- Use methods of the **DirectMusicComposer8** class to create a composition or transition at run time. See Overview of Programming for Composition and Using Transitions.
- Make a copy of an existing segment by using the **DirectMusicSegment8.Clone** method.
- Use the **DirectMusicBand8.CreateSegment** method. This creates a special type of secondary segment that is used only for making band changes. See Making Band Changes Programmatically.

Segments can perform different roles in the performance. There must always be a *primary segment*, which provides the main content of the soundtrack and normally serves as the control segment. S*econdary segments* play along with the primary segment and might provide sound effects or short musical themes. A special type of secondary segment is the *motif*, which is always obtained from a DirectMusic style object.

In addition there are three kinds of segments with special roles:

- *Transition segment*. A short musical transition created at run time by the DirectMusic composer object and normally played as a primary segment leading from one segment to another, or from a segment to silence.

- *Band segment*. A set of instruments and instrument settings for the various channels in the performance. The application can play a band segment as a secondary segment to execute changes in the band performing the music.
- *Template segment*. A guide to chord progressions, groove levels, and embellishments, used in conjunction with a style and chordmap to compose music at run time.

[C++]
The playback of segments is controlled by the performance object and begins with a call to **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx**.

[Visual Basic]
The playback of segments is controlled by the **DirectMusicPerformance8** object and begins with a call to **DirectMusicPerformance8.PlaySegmentEx**.

Only one primary segment can play inside a performance. When you cue a primary segment for playback, you can specify that it is to be played after the currently playing segment is finished, or you can use it to replace the current primary segment.

Secondary segments, on the other hand, play over the current primary segment, and any number of secondary segments can be playing simultaneously.

Secondary segments do not normally alter the performance of the primary segment. For example, a secondary segment can be based on a different style without affecting the style of the primary segment. However, a secondary segment can be designated as the control segment, in which case it takes over certain tasks normally handled by the primary segment.

More information on segments is contained in the following topics:

- Segment States
- Control Segments
- Self-Controlling Segments
- Wave Segments

See also Segment Timing.

Segment States

[C++]
When you play a segment, parameters for that segment such as the audiopath, repetitions, and start point are stored in a segment state, represented by an **IDirectMusicSegmentState8** interface. The parameters are valid only for that instance of the segment. Changes subsequently made to the segment by using

methods of **IDirectMusicSegment8** are reflected in new segment states created by calls to **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx**.

When different instances of a segment are being played on different audiopaths, you can use the segment state to retrieve a 3-D sound buffer or an effect, and make changes that apply only to that instance. For example, you might use the same engine sound for different cars in a race game, playing the sound for each car on its own audiopath. You can use either **IDirectMusicAudioPath8::GetObjectInPath** or **IDirectMusicSegmentState8::GetObjectInPath** to retrieve an **IDirectSound3DBuffer8** interface from each audiopath, allowing you to set the 3-D parameters for each car individually.

[Visual Basic]
When you play a segment, parameters for that segment such as the audiopath, repetitions, and start point are stored in a segment state object, represented by a **DirectMusicSegmentState8** object. The parameters are valid only for that instance of the segment. Changes subsequently made to the segment by using methods of **DirectMusicSegment8** are reflected in new segment states created by calls to **DirectMusicPerformance8.PlaySegmentEx**.

Control Segments

[C++]
The control segment is the source of any data that is shared across tracks. The following tracks give their data to the performance not by sending messages, as most other tracks do, but by responding to parameter calls.

- Chord
- Command
- Mute
- Tempo
- Time signature

The chord track, for example, answers parameter calls from the style track. To determine the MIDI value for a note before sending that note, the style track must determine the current chord. It does so by calling **IDirectMusicPerformance8::GetParam**, and this call is relayed to the chord track in the control segment.

To function as a control segment, a segment must have at least one controlling track. The mute, command, tempo, and chord tracks are controlling tracks.

The control segment does not affect any aspect of playback that is controlled by messages. The time signature comes from the control segment only when there is no time signature track, as is normally the case in segments not based on MIDI files.

By default, the primary segment is the control segment. However, a secondary segment can be designated the control segment by passing the DMUS_SEGF_CONTROL flag to **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx**.

When a secondary segment is the control segment, the primary segment continues to function as a fallback source of control data. For example, if a secondary control segment does not contain a tempo track, but the primary segment does, the tempo comes from the primary segment.

For more information, see the following topics:

- **DMUS_SEGF_FLAGS**
- Track Configuration

[Visual Basic]
A secondary segment can be designated a *control segment*, in which case it takes over the task of responding to certain calls such as **DirectMusicPerformance8.GetCommand** and controls some aspects of playback, such as the current tempo, chord, and groove level.

By default, the primary segment is the control segment. However, a secondary segment can be designated the control segment by passing the DMUS_SEGF_CONTROL flag to **DirectMusicPerformance8.PlaySegmentEx**. For more information, see **CONST_DMUS_SEGF_FLAGS**.

When a secondary segment is the control segment, the primary segment continues to function as a fallback source of control data. For example, if a secondary control segment does not contain a tempo track but the primary segment does, the tempo comes from the primary segment.

Self-Controlling Segments

A self-controlling segment ignores any control information from the control segment that duplicates control information in the self-controlling segment. For example, if the segment has a command track, it can use its own groove levels rather than the groove levels set by the control segment.

Segments can define for each track where it gets its controlling information, as follows:

- From tracks in the control segment. This is the default behavior.

- From tracks in the primary segment, regardless of whether it is the control segment.
- From tracks in the same segment.

[C++]
Segments are usually configured to be self-controlling by the author. However, applications can configure individual tracks within segments by setting or clearing the following flags, using the **IDirectMusicSegment8::SetTrackConfig** or **IDirectMusicSegmentState8::SetTrackConfig** method:

DMUS_TRACKCONFIG_OVERRIDE_ALL
: The track should get parameters from this segment before control and primary tracks.

DMUS_TRACKCONFIG_OVERRIDE_PRIMARY
: The track should get parameters from this segment before the primary segment tracks.

DMUS_TRACKCONFIG_FALLBACK
: The track should get parameters from this segment if the primary and control segments don't return the needed information.

The following example code, where *pSegment* is an **IDirectMusicSegment8** interface pointer, instructs the style track to get all its parameters from other tracks in the same segment, ensuring that chords, groove levels, and mute commands do not come from the control segment.

```
HRESULT hr = pSegment ->SetTrackConfig(CLSID_DirectMusicStyleTrack,
    -1, DMUS_SEG_ALLTRACKS, DMUS_TRACKCONFIG_OVERRIDE_ALL, 0);
```

For more information, see Track Configuration.

[Visual Basic]
Segments are configured to be self-controlling by the author. DirectX for Visual Basic does not support track configuration by applications.

Wave Segments

The DirectMusic loader can load compressed wave-form audio files in any format supported by the audio compression manager (ACM), as well as uncompressed wave files.

[C++]
Segments based on wave files are played just like any other segment. They pass through the performance as **DMUS_WAVE_PMSG** messages and are always played on channel 0 of the audiopath. Although waves are not synthesized in the same sense

as musical notes, they do pass through the synthesizer and can be manipulated by MIDI controllers.

[Visual Basic]
Segments based on wave files are played just like any other segment. They pass through the performance as messages and are always played on channel 0 of the audiopath. Although waves are not synthesized in the same sense as musical notes, they do pass through the synthesizer and can be manipulated by MIDI controllers.

The following example code, where *dmp* is a **DirectMusicPerformance8** object, sends a pitch change MIDI message to channel 0 on the default audiopath. The change in pitch affects every wave played on that audiopath.

```
Private Sub sldrPitch_Change()

    Dim hi As Byte, lo As Byte

    ' Split value into 7-bit bytes. The value returned by the slider
    ' is in the range 0 to 16383.

    hi = Fix(sldrPitch.Value / 128)
    lo = CByte(sldrPitch.Value And 127)

    ' Send pitch bend message.

    Call dmp.SendMIDIPMSG(0, DMUS_PMSGF_REFTIME, 0, &HE0, _
       lo, hi)

End Sub
```

Note

The technique shown in the example is not guaranteed to work if your application uses multiple audiopaths. In DirectX for Visual Basic, there is no way to direct an application-created message to a particular audiopath. Because performance channel 0 is mapped to different channels on the synthesizer by different audiopaths, a MIDI controller change on one path has no effect on other paths. For more information, see Application-Created Messages.

Waves can also be embedded in DirectMusic Producer files, where they can behave somewhat like styles with variations. For an example, see the AudioScripts sample application, where the ScriptDemoBaseball.spt script plays variations on the vendor's cry each time a button is clicked.

Waves in segment (.sgt) files must be downloaded to the synthesizer before being played. They are downloaded when the segment's bands are downloaded. For more information, see Downloading and Unloading Bands.

Using Bands

A band is a choice of instruments assigned to particular parts in a segment or style. At performance time, each instrument track is mapped to a performance channel, which stores the following information:

- MIDI volume.
- MIDI pan.
- Transposition. If this value is nonzero, music notes on the channel are automatically transposed for the instrument.
- The instrument's MIDI patch number, including MSB and LSB bank selects.
- A reference to the DLS collection from which to load the instrument. By default, the DLS collection is the standard General MIDI collection.

Segments and styles always contain at least one band, called the default band. Styles can contain additional bands. When you load a segment or style, the default band and any other bands are automatically loaded as well. However, you must still download the DLS data for the instruments in any band that you intend to use.

[C++]
You can retrieve a pointer to the default band by using the **IDirectMusicStyle8::GetDefaultBand** method.

Other bands might be authored into the style, and can be found and retrieved by using the **IDirectMusicStyle8::EnumBand** and **IDirectMusicStyle8::GetBand** methods. Bands can also be obtained from other style files or from band files. Once you have obtained an **IDirectMusicBand8** interface, you have access to that band and can substitute it for the default band.

[Visual Basic]
You can retrieve an object representing the default band by using the **DirectMusicStyle8.GetDefaultBand** method. Other bands might be authored into the style and can be retrieved by using the **DirectMusicStyle8.GetBand** method. Bands can also be obtained from other style files or from band files. Once you have obtained a **DirectMusicBand8** object, you have access to that band and can substitute it for the default band.

Bands are loaded like other objects. For more information, see Loading Audio Data.

More information about bands is contained in the following topics:

- Downloading and Unloading Bands
- Making Band Changes Programmatically
- Ensuring Timely Band Changes
- Playing a MIDI File with Custom Instruments

Downloading and Unloading Bands

Before a band can be used, the instruments it references must be downloaded to the performance. This step maps the instruments to performance channels and downloads the DLS data to the synthesizer.

By default, the application is responsible for downloading any band it uses. However, you can turn on automatic downloading of bands.

Downloading a band makes the band available to the performance but does not perform any program changes. Program changes take place in response to messages generated by the segment's band track, which is typically authored into a segment file. For information on how to make program changes at run time, see Making Band Changes Programmatically.

Information about how to implement downloading and unloading of bands is contained in the following topics:

- Automatically Downloading Bands
- Manually Downloading Bands
- Patch Collisions
- Unloading Bands

Automatically Downloading Bands

When automatic downloading is on, the instruments in the band are downloaded when the segment containing the band is cued. The instruments are automatically unloaded when the segment is stopped, unless another segment using the same instruments is cued to play immediately or is currently playing.

Automatic downloading should be used only when the timing of segment starts is not critical. Repeated loading and unloading of instruments takes time and can cause serious degradation of performance in complex audio environments.

Automatic unloading, which is part of the automatic downloading mechanism, can also lead to undesired results. For example, suppose you play a short secondary segment that changes the instrument on a channel. The instrument is automatically downloaded when the secondary segment starts, replacing the existing instrument. When the secondary segment ends, the instrument is automatically unloaded, with the result that there is no instrument on that channel, and the channel plays silence.

[C++]
You can turn on automatic downloading of bands in one of the following ways:

- Call the **IDirectMusicPerformance8::SetGlobalParam** method for the GUID_PerfAutoDownload parameter. See Setting and Retrieving Global Parameters and the following example.
- Enable automatic downloading for a single segment by calling the **IDirectMusicSegment8::SetParam** method for the GUID_Enable_Auto_Download parameter. See Setting and Retrieving Track Parameters.

In the following code example, the global parameter for the performance *pPerf* is set to enable automatic downloading of bands:

```
BOOL fAuto = TRUE;
pPerf->SetGlobalParam(GUID_PerfAutoDownload, &fAuto, sizeof(BOOL));
```

[Visual Basic]
You can turn on automatic downloading of bands for the entire performance by using the **DirectMusicPerformance8.SetMasterAutoDownload** method, or for an individual segment by using the **DirectMusicSegment8.SetAutoDownloadEnable** method.

Manually Downloading Bands

[C++]
You can manually download a band in one of the following ways:

- Call **IDirectMusicSegment8::Download** to download the bands and waves in a segment to either an audiopath or a performance.
- Obtain an **IDirectMusicBand8** interface from a loaded object, and call the **IDirectMusicBand8::Download** method.
- Call the **IDirectMusicSegment8::SetParam** method with the GUID_Download or GUID_DownloadToAudioPath parameter to download the band in the segment's first band track. You can also use **IDirectMusicPerformance8::SetParam** to set this parameter on the primary segment, or **IDirectMusicTrack8::SetParamEx** to set it directly on a band track. See Setting and Retrieving Track Parameters.

If your application creates audiopaths that use more than one synthesizer, or port, you must download bands to the individual audiopaths, not to the performance. However, most applications use only a single synthesizer, and it is safe to download all instrument data to the performance.

Multiple audiopaths usually share the same synthesizer. When a band is downloaded to an audiopath, the instrument data is downloaded to the port on that audiopath and

is then available to any audiopath using the same port. The data does not have to be downloaded to each audiopath.

For example, suppose you have a segment that uses a DLS instrument for the sound of a car engine in a race game. To manipulate the 3-D parameters independently for each car in the race, you play the sound on multiple audiopaths. The instrument has to be downloaded only once.

There is no danger in downloading the same instrument multiple times. If an instrument appears in one band multiple times or if it appears in multiple bands that are all opened and downloaded at the same time, only one copy of the instrument is sent to the synthesizer.

The following function loads a band from disk and downloads it to the performance:

```
HRESULT myDownloadBand(
    IDirectMusicLoader8 *pILoader,      // Loader interface.
    IDirectMusicBand8 **ppBand,         // To retrieve pointer.
    IDirectMusicPerformance8 *pPerf,    // Performance to use band.
    WCHAR *pwszFile)                    // File to load.
{
    HRESULT hr;
    DMUS_OBJECTDESC Desc;

    // Start by initializing Desc with the file name
    // and GUID for the band object.

    wcscpy(Desc.wszFileName,pwszFile);
    Desc.dwSize = sizeof(Desc);
    Desc.guidClass = CLSID_DirectMusicBand;
    Desc.dwValidData = DMUS_OBJ_CLASS |
          DMUS_OBJ_FILENAME | DMUS_OBJ_FULLPATH;

    hr = pILoader->GetObject(&Desc, IID_IDirectMusicBand8,
          (void **) ppBand);
    if (SUCCEEDED(hr))
    {
       hr = (*ppBand)->Download(pPerf);
    }
    return hr;
}
```

[Visual Basic]
You can manually download a band in one of the following ways:

- Call **DirectMusicSegment8.Download** to download the bands in a segment to either an audiopath or a performance.

- Retrieve a **DirectMusicBand8** object from a file, a resource, or a **DirectMusicStyle8** object and then call its **DirectMusicBand8.Download** method.

Patch Collisions

When bands from different segments are downloaded to the same port, instruments in band can overwrite data from a previously downloaded band.

[C++]
For example, suppose segment A uses a band that assigns a piano to patch number 1, and segment B uses a band that assigns a banjo to the same patch. The application calls **IDirectMusicSegment8::Download** first for segment A and then for segment B. Even though the bands might be downloaded to different audiopaths, the instrument data is downloaded to the same synthesizer, so any note on a performance channel mapped to patch number 1 will be played by the banjo.

[Visual Basic]
For example, suppose segment A uses a band that assigns a piano to patch number 1, and segment B uses a band that assigns a banjo to the same patch. The application calls **DirectMusicSegment8.Download** first for segment A and then for segment B. Even though the bands might be downloaded to different audiopaths, the instrument data is downloaded to the same synthesizer, so any note on a performance channel mapped to patch number 1 will be played by the banjo.

This potential for patch collisions must be taken into account when the content is authored. Different segments should not use different instruments with the same patch number.

Unloading Bands

[C++]
Bands take up memory, so they should be unloaded when they are no longer in use. If you have enabled automatic downloading of bands, the bands associated with a segment are unloaded automatically when the segment stops. Otherwise, you can manually unload a band in one of the following ways:

- Call the **IDirectMusicSegment8::Unload** or **IDirectMusicBand8::Unload** method for instruments downloaded by the corresponding **Download** method.
- Call the **IDirectMusicSegment8::SetParam** method for the GUID_Unload or GUID_UnloadFromAudioPath parameter to unload the band in the segment's band track. You can also use **IDirectMusicPerformance8::SetParam** to set this parameter on the primary segment, or **IDirectMusicTrack8::SetParamEx** to set it directly on a band track. See Setting and Retrieving Track Parameters.

The **IDirectMusicPerformance8::CloseDown** method also unloads any remaining downloaded instruments.

[Visual Basic]
Bands take up memory, so they should be unloaded when they are no longer in use. If you have enabled automatic downloading of bands, the bands associated with a segment are unloaded automatically when the segment stops. Otherwise, you can manually unload a band by calling the **DirectMusicSegment8.Unload** or **DirectMusicBand8.Unload** method for instruments downloaded by the corresponding **Download** method.

The **DirectMusicPerformance8.CloseDown** method also unloads any remaining downloaded instruments.

Making Band Changes Programmatically

Usually, the band track in a loaded segment will perform program changes. However, you can also do so manually.

[C++]
First create a secondary segment with a call to the **IDirectMusicBand8::CreateSegment** method, and then play that segment by calling **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx**. Typically, you would use DMUS_SEGF_MEASURE or DMUS_SEGF_GRID in the *dwFlags* parameter to ensure that the band change takes effect on an appropriate boundary.

The following sample function creates a segment from a band and plays it. It is presumed that the instruments have been downloaded or that automatic downloading has been enabled.

```
HRESULT myPlayBand(
    IDirectMusicBand8 *pBand,          // Pointer to a band object.
    IDirectMusicPerformance8 *pPerf,   // Performance to use the band.
    REFERENCE_TIME rfTime,             // Time to play at.
    DWORD dwFlags)                     // Performance flags.
{
    IDirectMusicSegment *pSegment;

  // We don't need IDirectMusicSegment8, which we'd have to QI for.

    HRESULT hr = pBand->CreateSegment(&pSegment);
    if (SUCCEEDED(hr))
    {
        hr = pPerf->PlaySegment(pSegment,
```

```
            dwFlags | DMUS_SEGF_SECONDARY,
            rfTime,
            NULL);
        pSegment->Release();
    }
    return hr;
}
```

[Visual Basic]
First create a segment by using **DirectMusicBand8.CreateSegment**, and then play that segment by calling **DirectMusicPerformance8.PlaySegmentEx**. Typically, you would use DMUS_SEGF_MEASURE or DMUS_SEGF_GRID in the *lFlags* parameter to ensure that the band change takes effect on an appropriate boundary.

A performance can be playing instruments from more than one band at a time. For example, suppose your application is playing a primary segment using one band, and then plays a motif from a style that has a different band. As long as the instruments in the two bands are mapped to different performance channels, no conflict arises. Note, though, that motif segments don't always have their own band tracks, so you might get silence from the motif's channels unless you first create a band segment and play it.

Ensuring Timely Band Changes

A consideration in playing band segments is the randomness in the timing of notes played by a style track. For instance, a note that is on measure 1, beat 1 might actually play somewhat earlier or later than the actual beat boundary. The band segment isn't aware of this, with the result that some of the notes might play with the incorrect instrument.

[C++]
To prevent this problem, an application should cue the band segment early. Suppose, for example, that you have a style segment *pStyleSeg* and a band segment *pBandSeg*. You want to play both the style segment and the band segment on the next measure boundary of the performance (*pPerf*). You know that the style contains notes that could go out up to 30 ticks earlier, in music time, than the start time of the segment. The following code example ensures that the band segment is played 31 ticks before the style segment, so all instruments are in place before any note is played:

```
// First get the time of the next measure, and convert it to
// music time.

REFERENCE_TIME rtResolved;
MUSIC_TIME mtResolved;
```

```
pPerf->GetResolvedTime( 0, &rtResolved, DMUS_TIME_RESOLVE_MEASURE );
pPerf->ReferenceToMusicTime( rtResolved, &mtResolved );

// Now play the band segment 31 ticks before the measure boundary.

mtResolved -= 31;
pPerf->PlaySegment(pBandSeg, 0, mtResolved, NULL);

// Play the style segment on the measure boundary.

pPerf->PlaySegment(pStyleSeg, DMUS_TIME_RESOLVE_MEASURE, 0, NULL);
```

[Visual Basic]

To prevent this problem, an application should cue the band segment early. Suppose, for example, that you have a style segment *styleSeg* and a band segment *bandSeg*. You want to play both the style segment and the band segment on the next measure boundary of the performance (*perf*). You know that the style contains notes that could go out up to 30 ticks earlier (in music time) than the start time of the segment. The following code example ensures that the band segment is played 31 ticks before the style segment, so all instruments are in place before any note is played:

```
'First get the time of the next measure
' and convert it to music time.

Dim ctResolved As Long
Dim mtResolved As Long

ctResolved = perf.GetResolvedTime(ctResolved, DMUS_SEGF_MEASURE)
mtResolved = perf.ClockToMusicTime(ctResolved)

' Now play the band segment 31 ticks before the measure boundary.

mtResolved = mtResolved - 31
Call perf.PlaySegmentEx(bandSeg, 0, mtResolved)

' Play the style segment on the measure boundary.

Call perf.PlaySegmentEx(styleSeg, DMUS_SEGF_MEASURE, 0 )
```

Note

If there is no randomness in the notes played by a segment (for example, one loaded from a MIDI file), you don't need to worry about the timeliness of a band segment played at the same time. By default, all band segments start 1 tick early.

Playing a MIDI File with Custom Instruments

By default, when you play a MIDI file the instruments used are those in the Roland GM/GS Sound Set, contained in the Gm.dls file. However, you can use instruments from any DLS collection when playing a MIDI file.

[C++]
First load the collection and retrieve a pointer to the **IDirectMusicCollection8** interface.

The following example function loads a collection by file name:

```
HRESULT LoadCollectionByName(
      IDirectMusicLoader8 *pILoader,
      char *pszFileName,
      IDirectMusicCollection8 **ppICollection)
{
   HRESULT hr;
   DMUS_OBJECTDESC Desc;          // Descriptor.

   // Start by initializing Desc with the file name
   // and GUIDfor the collection object.
   // The file name starts as a char string, so convert
   // to Unicode.

   mbstowcs(Desc.wszFileName,pszFileName,DMUS_MAX_FILENAME);
   Desc.dwSize = sizeof(DMUS_OBJECTDESC);
   Desc.guidClass = CLSID_DirectMusicCollection;
   Desc.dwValidData = DMUS_OBJ_CLASS
         | DMUS_OBJ_FILENAME
         | DMUS_OBJ_FULLPATH;

   hr = pILoader->GetObject(&Desc,
         IID_IDirectMusicCollection8,
         (void **) ppICollection);
   return hr;
}
```

Next you must connect the DLS data to the segment by calling **IDirectMusicSegment8::SetParam**.The following code example shows how to make a connection between a segment and a collection returned by the LoadCollectionByName function.

```
// Assume that pSegment was created from a MIDI file and that
```

```
// pLoadedCollection is a valid IDirectMusicCollection8 pointer.

HRESULT hr = pSegment->SetParam(GUID_ConnectToDLSCollection,
    0xFFFFFFFF, 0, 0,
    (void*)pLoadedCollection);
```

Finally, download the instruments in the collection to the performance or audiopath by calling **IDirectMusicSegment8::Download**.

[Visual Basic]
To do so, load the collection, associate it with the segment based on the MIDI file, and download the collection as you would with any other segment.

The following example code, where *dmLoader* is a **DirectMusicLoader8** object, loads a DLS collection, connects it to the **DirectMusicSegment8** object represented by *dmSeg*, and downloads it to the default audio path of the **DirectMusicPerformance8** object *dmPerf*:

```
Dim dmColl As DirectMusicCollection8

Set dmColl = dmLoader.LoadCollection("boids.dls")
dmSeg.ConnectToCollection dmColl
dmSeg.Download dmPerf.GetDefaultAudioPath
```

When a custom collection is attached to a MIDI segment, the connection to the GM collection is not broken. For example, suppose you load a collection containing a single instrument that has a patch number of 12 and connect this to the segment. MIDI channels with any patch number other than 12 continue to be played by the appropriate instruments in the GM collection.

For more information on collections, see Using Instrument Collections.

Timing

This section is an overview of various timing issues in DirectMusic. The following topics are discussed:

- Master Clock
- Clock Time vs. Music Time
- Changing the Tempo
- Prepare Time
- Latency and Bumper Time
- Segment Timing

Master Clock

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not enable applications to select a different master clock.

[C++]
To guarantee accurate timing with an acceptably low latency, DirectMusic incorporates a master clock in kernel mode. This clock is based on a hardware timer. DirectMusic automatically selects the system clock as the master clock, but an application can select a different one, such as the wave-out crystal on a sound card.

The master clock is a high-resolution timer that is shared by all processes, devices, and applications that are using DirectMusic. The clock is used to synchronize all audio playback in the system. It is a standard **IReferenceClock** interface. The **IReferenceClock::GetTime** method returns the current time as a 64-bit integer (defined as the **REFERENCE_TIME** type) in increments of 100 nanoseconds.

To obtain an interface to the master clock, call the **IDirectMusic8::GetMasterClock** method.

You can choose a different master clock for your application, but only if there are no other DirectMusic applications running. First you get descriptions of all devices that can serve as the master clock by using the **IDirectMusic8::EnumMasterClock** method. Once you have obtained the GUID of the device that you want to use as the master clock, you pass this to the **IDirectMusic8::SetMasterClock** method.

Clock Time vs. Music Time

[C++]
In DirectX for C++, the time returned by the master clock is a 64-bit value defined as type **REFERENCE_TIME**. Reference time is measured in units of approximately 100 nanoseconds, so the clock ticks about 10 million times each second. The value returned by the **IReferenceClock::GetTime** method is relative to an arbitrary start time.

Music time is a 32-bit value defined as type **MUSIC_TIME**. It is not an absolute measure of time but is relative to the tempo. The clock is started when the performance is initialized and ticks DMUS_PPQ times for each quarter-note. DMUS_PPQ is defined as 768.

When a performance is initialized, it starts keeping an internal clock. You can retrieve the current performance time in both reference time and music time by using the **IDirectMusicPerformance8::GetTime** method.

The **IDirectMusicPerformance8::AdjustTime** method can be used to make small changes to the performance time. Most applications don't need to do this, but it can be useful when synchronizing to another source.

To convert between the two kinds of time in a performance, you can use the **IDirectMusicPerformance8::MusicToReferenceTime** and **IDirectMusicPerformance8::ReferenceToMusicTime** methods. These methods convert between time offsets within the performance, taking into account all tempo changes that have taken place since the performance started.

When a segment is cued to play by a call to **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx** and the start time is given in reference time, DirectMusic must convert the start time to music time. If no primary segment is currently playing, the conversion is made immediately, based on the current tempo. Otherwise, if another segment is playing, the start time of the cued segment is not converted to music time until the start time has been reached.

If the tempo is changed before the segment starts playing, the actual start time can be affected, or the segment might not start on the desired boundary. In the first case, in which the conversion to music time is done immediately, the start time (in reference time) is advanced if the tempo speeds up and delayed if the tempo slows down. In the second case, in which conversion is made at start time, a change in tempo can mean that the segment does not start at correct resolution boundaries. For example, if the segment is supposed to start on a measure boundary (as indicated in the *dwFlags* parameter of **PlaySegment** or **PlaySegmentEx**), the actual start time (in reference time) is calculated when the segment is cued. However, if the tempo then changes, a measure boundary might not fall at that time.

When a primary segment is played with the DMUS_SEGF_QUEUE flag (see **DMUS_SEGF_FLAGS**), the *i64StartTime* parameter is ignored, and the segment is cued to play after any primary segments whose start times have already been converted. If a previously cued segment is still stamped in reference time, that segment will play at its designated time, perhaps interrupting another segment.

[Visual Basic]
In DirectX for Visual Basic, the time returned by the master clock is a **Long** representing units of approximately one millisecond. The value returned by **DirectMusicPerformance8.GetClockTime** is relative to an arbitrary start point.

Music time is also a **Long**. It is not an absolute measure of time but is relative to the tempo. The clock is started when the performance is initialized and ticks 768 times for each quarter-note. You can retrieve the current music time by using **DirectMusicPerformance8.GetMusicTime**.

To convert between the two kinds of time in a performance, you can use the **DirectMusicPerformance8.MusicToClockTime** and **DirectMusicPerformance8.ClockToMusicTime** methods.

When a segment is cued to play by a call to **DirectMusicPerformance8.PlaySegmentEx** and the start time is given in clock time, DirectMusic must convert the start time to music time. If no primary segment is currently playing, the conversion is made immediately, based on the current tempo. Otherwise, if another segment is playing, the start time of the cued segment isn't converted to music time until the start time has been reached.

If the tempo is changed before the segment starts playing, the start time can be affected, or the segment might not start on the desired boundary. In the first case, in which the conversion to music time is made immediately, the start time (in reference time) is advanced if the tempo speeds up and delayed if the tempo slows down. In the second case, in which conversion is made at start time, a change in tempo can mean that the segment does not start at correct resolution boundaries. For example, if the segment is supposed to start on a measure boundary (as indicated in the *lFlags* parameter of **PlaySegmentEx**), the actual start time (in clock time) is calculated when the segment is cued. However, if the tempo then changes, a measure boundary might not fall at that time.

When a primary segment is passed to **PlaySegmentEx** with the DMUS_SEGF_QUEUE flag (see **CONST_DMUS_SEGF_FLAGS**), the *startTime* parameter is ignored and the segment is cued to play after any primary segments whose start times have already been converted. If a previously cued segment is still stamped in reference time, that segment plays at its designated time, perhaps interrupting another segment.

For example, say you have three segments, each 10 seconds in length. You cue segment A to play 5 seconds from now. Because no primary segment is currently playing, the start time is immediately converted to music time. At 6 seconds, you cue segment B to play at 20 seconds. In this case, because music is already playing and the tempo might change, the conversion to music time is not made immediately. Then you cue segment C with the DMUS_SEGF_QUEUE flag so that it starts immediately after segment A finishes, at 15 seconds. At 20 seconds, segment B starts playing and interrupts segment C.

Changing the Tempo

The tempo of a performance dictates the conversion between the two types of time used in DirectMusic, which in turn controls the resolution of events to musical boundaries. (See Clock Time vs. Music Time.) The tempo track of the control segment usually manages the tempo, but it is also possible for an application to set the tempo dynamically.

[C++]
There are two ways to change the tempo: by sending a message and by setting a track parameter on the control segment.

The following code example sends a message to change the tempo:

```
// Assume that pIDMSegment is a valid IDirectMusicSegment8 and
// IDMPerformance is a valid IDirectMusicPerformance8.

// Disable tempo track in segment so that it does not reset the tempo.

pIDMSegment->SetParam( GUID_DisableTempo, 0xFFFF,0,0, NULL );

DMUS_TEMPO_PMSG* pTempo;

if( SUCCEEDED(pIDMPerformance->AllocPMsg(
      sizeof(DMUS_TEMPO_PMSG), (DMUS_PMSG**)&pTempo)))
{
   // Cue the tempo event.
   ZeroMemory(pTempo, sizeof(DMUS_TEMPO_PMSG));
   pTempo->dwSize = sizeof(DMUS_TEMPO_PMSG);
   pTempo->dblTempo = 100;
   pTempo->dwFlags = DMUS_PMSGF_REFTIME;
   pTempo->dwType = DMUS_PMSGT_TEMPO;
   pIDMPerformance->SendPMsg((DMUS_PMSG*)pTempo);
}
```

The following example shows how to change the tempo parameter. For more information, see Setting and Retrieving Track Parameters.

```
DMUS_TEMPO_PARAM Tempo;
Tempo.dblTempo = 100;
pIDMSegment->SetParam(GUID_TempoParam, 0xFFFF, 0, 0, &Tempo);
```

You can also change the master tempo, which adjusts the tempo set by any control segment. See Setting and Retrieving Global Parameters

---

[Visual Basic]
There are two ways to do so: by setting the master tempo and by sending a tempo message.

The master tempo is a factor by which all tempos in the performance are multiplied. For example, if you set the master tempo by calling **DirectMusicPerformance8.SetMasterTempo** with a parameter of 0.75, and then play a segment that has a tempo of 120 beats per minute, the segment plays with a tempo of 90.

Sending a tempo by using the **DirectMusicPerformance8.SendTempoPMSG** method changes the tempo at the time for which the message is stamped. The new tempo is valid until another tempo message is sent or the tempo is changed by a control segment. The tempo value can be modified by the master tempo.

The following call, where *perf* is a **DirectMusicPerformance8** object, immediately sets the tempo to 100 beats per minute. Note that if you pass 0 as the *lTime* parameter to signify that the message is to go out immediately, you must also set the DMUS_PMSGF_REFTIME flag.

```
Call perf.SendTempoPMSG(0, DMUS_PMSGF_REFTIME, 100)
```

Prepare Time

As a segment is played, the performance makes repeated calls to the segment's tracks, causing them to generate messages for the supplied time range, which is some fraction of a second. These messages are then placed in the queue behind those that were generated in previous calls. By default, about a second's worth of messages are in the queue at any given time.

Each time the performance calls on a track to play messages, it calculates the end time for that call by adding the prepare time to the current time. If the current time is 10,000 milliseconds (or the equivalent in REFERENCE_TIME units) and the prepare time is the default 1000 ms, the end time is 11,000—that is, all new messages that are to play up to time 11,000 must be prepared and placed in the queue.

[C++]
The size of the queue can be changed by calling the **IDirectMusicPerformance8::SetPrepareTime** method, and the current size can be retrieved by using **IDirectMusicPerformance8::GetPrepareTime**.

Most applications don't need to change the default prepare time, and the process just described is not visible to the application. However, it is helpful to understand the concept of prepare time because of the DMUS_SEGF_AFTERPREPARETIME flag, which the application can pass to **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx**.

[Visual Basic]
The size of the queue can be changed by calling the **DirectMusicPerformance8.SetPrepareTime** method, and the current size can be retrieved by using **DirectMusicPerformance8.GetPrepareTime**.

Most applications don't need to change the default prepare time, and the process just described is not visible to the application. However, it is helpful to understand the concept of prepare time because of the DMUS_SEGF_AFTERPREPARETIME flag, which the application can pass to **DirectMusicPerformance8.PlaySegmentEx**.

Without this flag, if you set a start time of "now" for the segment, the performance invalidates any messages currently in the queue. Any tracks that are still valid at this

point—for example, tracks of secondary segments that continue to play despite the introduction of a new primary segment—then have to resend their messages, taking into account any changes made to the environment by the new segment. This causes extra processing and might also lead to undesired results.

You can use the DMUS_SEGF_AFTERPREPARETIME flag to specify that the segment isn't to start playing until all messages currently in the queue have been processed and passed to the port buffer. If messages up to time 10,000 milliseconds (ms) are in the queue and the current time is 9,000 ms, a segment cued to play immediately, but flagged DMUS_SEGF_AFTERPREPARETIME, starts playing just after the 10,000 ms mark.

For more information, see Segment Timing. For an illustration, see Latency and Bumper Time.

Latency and Bumper Time

[C++]
Latency is the delay between the time at which the port receives a message and the time at which it has synthesized enough of a wave to play. The **IDirectMusicPerformance8::GetLatencyTime** method retrieves the current time plus the latency for the performance as a whole. This is the largest value returned by any of the ports' latency clocks.

The bumper is an extra amount of time allotted for code to run between the time that an event is put into the port buffer and the time that the port starts to process it. By default, the bumper length is 50 milliseconds. An application can change this value by using the **IDirectMusicPerformance8::SetBumperLength** method, and retrieve the current value by calling **IDirectMusicPerformance8::GetBumperLength**.

Here's an example of how latency time and bumper time are combined. Suppose an event is supposed to play at 10,000 milliseconds (ms). The latency of the port is known to be 100 ms, and the bumper length is at its default value of 50 ms. The performance therefore places the message into the port buffer at 9,850 ms.

Any tools that alter the time of messages must take latency and bumper time into account. If a tool stamps a message with a time that is already past the latency time, the note or other event will not play at the correct time.

Once a message has been placed in the port buffer, it no longer belongs to the performance and cannot be stopped from playing by using the **IDirectMusicPerformance8::Invalidate** method or by stopping the segment. The first message that can be invalidated has a time stamp equal to or greater than the current time plus the latency time and the bumper time. This value can be retrieved by using the **IDirectMusicPerformance8::GetQueueTime** method.

[Visual Basic]
Latency is the delay between the time at which the port receives a message and the time at which it has synthesized enough of a wave to play. The

DirectMusicPerformance8.GetLatencyTime method retrieves the current time plus the latency for the performance as a whole. This is the largest value returned by any of the ports' latency clocks.

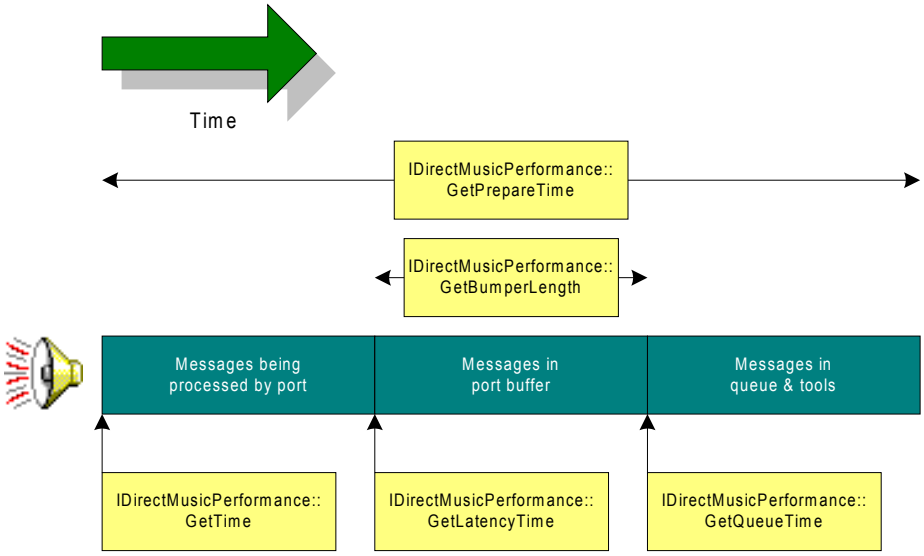
The bumper is an extra amount of time allotted for code to run between the time that an event is put into the port buffer and the time that the port starts to process it. By default, the bumper length is 50 milliseconds (ms). An application can change this value by using the **DirectMusicPerformance8.SetBumperLength** method, and retrieve the current value by calling **DirectMusicPerformance8.GetBumperLength**.

Here's an example of how latency time and bumper time are combined. Suppose an event is supposed to play at 10,000 ms. The latency of the port is known to be 100 ms, and the bumper length is at its default value of 50 ms. The performance therefore places the message into the port buffer at 9,850 ms.

Once a message has been placed in the port buffer, it no longer belongs to the performance and cannot be stopped from playing by using the **DirectMusicPerformance8.Invalidate** or the **DirectMusicPerformance8.StopEx** method. The first message that can be invalidated has a time stamp equal to or greater than the current time plus the latency and bumper time. This value can be retrieved by using the **DirectMusicPerformance8.GetQueueTime** method.

The following diagram, not to scale, illustrates the relationship of the times and durations retrieved by various methods. The current time is at the left, and the last time for which messages have been prepared is at the right. Remember that prepare time is only an approximation of the total timespan of messages in the queue at any given moment.

[Visual Basic]
The methods in the illustration are from the C++ API, but most have similar names in DirectX for Visual Basic. The **GetTime** method in the illustration is equivalent to **DirectMusicPerformance8.GetClockTime**.



Segment Timing

This section covers various aspects of timing in the playing of segments.

To clarify the relationship between times within segments and times within the performance, the following terms are used throughout this section:

Start point
The first point in the segment that can be a segment start time. By default this value is 0, indicating the beginning of the segment. However, it can be changed by the application.

Segment start time
The point in a segment where it begins producing sounds. This time is usually the same as the start point, but can be later if the start point is deliberately aligned to a play time that is in the past.

Play time
The point in the performance where a new segment's start point is cued. In the DirectMusic API, this time is sometimes called start time.

Start marker
A marker indicating a valid start time in a segment.

Play marker
A marker indicating a point in a control segment where another segment's start point can be cued.

[C++]

Note

Start markers and play markers are placed in the segment's marker track by the author and cannot be changed by the application. They can, however, be retrieved by using the GUID_Valid_Start_Time and GUID_Play_Marker parameters.

Segments normally play from the beginning. You can make a segment start from another point by using the **IDirectMusicSegment8::SetStartPoint** method. The new start point remains valid until changed.

If a repeat count is set by using **IDirectMusicSegment8::SetRepeats**, the entire segment repeats that number of times, unless a loop has been defined by a call to **IDirectMusicSegment8::SetLoopPoints**, in which case only the part of the segment between the loop points repeats.

The play time is determined by two parameters of the **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx** methods:

- The *i64StartTime* parameter sets the earliest time at which the segment can start playing. If *i64StartTime* is 0, this time is as soon as possible. The actual time at which the segment will start depends on the type of segment. If it is a primary segment or a secondary control segment, the earliest play time is at queue (or flush) time. If it is a noncontrol secondary segment, the earliest play time is at latency time. For more information on queue time and latency time, see Latency and Bumper Time.
- The *dwFlags* parameter specifies how soon after the earliest possible play time the segment will actually start playing. Usually, you will want to wait for an appropriate point in the rhythm before introducing a new primary segment, transition, or motif. You control the delay by setting one or more flags from the **DMUS_SEGF_FLAGS** enumeration.

[Visual Basic]

Note

Start markers and play markers are placed in the segment's marker track by the author and cannot be changed by the application.

Segments normally play from the beginning. You can make a segment start from another point by using the **DirectMusicSegment8.SetStartPoint** method. This start point is used whenever the segment is played.

If a repeat count is set by using **DirectMusicSegment8.SetRepeats**, the entire segment repeats that number of times, unless a loop has been defined by a call to **DirectMusicSegment8.SetLoopPoints**, in which case only the part of the segment between the loop points repeats.

The play time is determined by two parameters of **DirectMusicPerformance8.PlaySegmentEx**:

- The *startTime* parameter sets the earliest time at which the segment can start playing. If *startTime* is 0, this time is as soon as possible. The actual time at which the segment will play depends on the type of segment. If it is a primary segment or a secondary control segment, the earliest play time is at queue (or flush) time. If it is a noncontrol secondary segment, the earliest play time is at latency time. For more information on queue time and latency time, see Latency and Bumper Time.
- The *lFlags* parameter specifies how soon after the earliest possible play time the segment will actually start playing. Most often you will want to wait for an appropriate point in the rhythm before introducing a new primary segment, transition, or motif. You control the delay by setting oneor more flags from the **CONST_DMUS_SEGF_FLAGS** enumeration.

Aligning a Segment to a Past Time

Rather than forcing the segment start time to the next grid, beat, or measure in the control segment, you might want the segment to start playing sooner, yet still match the rhythm of the current segment. You can make the segment do so by cuing its start point to a rhythmic boundary that has already passed. The rhythm in the cued segment is thus aligned with that in the current segment, and the new segment can start playing immediately.

To cue the segment in the past, use the DMUS_SEGF_ALIGN flag. Add one of DMUS_SEGF_GRID, DMUS_SEGF_BEAT, or DMUS_SEGF_MEASURE to cue the start point of the segment at the appropriate rhythmic boundary. Alternatively, you can use DMUS_SEGF_MARKER to align the start point to the most recently played play marker in the control segment.

Of course, when the start point is in the past, the segment start time has to be adjusted to fall in the present or the future. The performance uses the following rules to determine the segment start time. In all cases, "next" means "next possible"—that is, within the part of the segment that does not fall in the past.

- If a start marker appears in the cued segment before the next resolution boundary of the specified type, the segment start time falls at that point.
- If there is no valid start marker, the segment start time is at the next start resolution boundary of the cued segment, as specified by one of the following flags:

  DMUS_SEGF_VALID_START_BEAT
  Put the segment start time on the next beat.

  DMUS_SEGF_VALID_START_GRID
  Put the segment start time on the next grid.

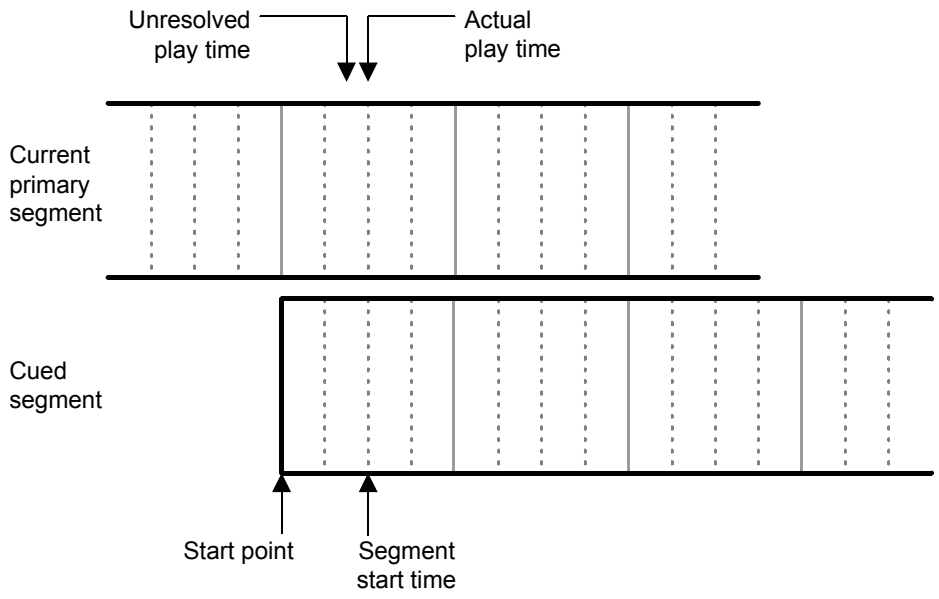
  DMUS_SEGF_VALID_START_TICK
  Put the segment start time at the earliest possible point.

- If there is no valid start marker and no start resolution flag is supplied, the segment start time is at the next play resolution boundary as specified by the DMUS_SEGF_GRID, DMUS_SEGF_BEAT, or DMUS_SEGF_MEASURE flag. If none of these flags is present, the segment start time is immediate.

Play markers and start markers allow greater flexibility in the cuing of segments, especially motifs. Suppose a motif is designed to sound best when it starts playing at the beginning of a measure in the primary segment. If the motif is cued with the DMUS_SEGF_MEASURE flag, there might be a significant delay before the next measure boundary is reached and the motif plays. But if the DMUS_SEGF_ALIGN flag is added, the motif can start playing sooner without violating the rhythm. Adding the DMUS_SEGF_MARKER flag ensures that the motif plays at an appropriate boundary within the control segment, rather than on just any measure, beat, or grid.

For information on how tempo changes can affect segment start times, see Clock Time vs. Music Time.

The following diagram shows how the timing is determined for a segment cued with the DMUS_SEGF_MEASURE, DMUS_SEGF_ALIGN, and DMUS_SEGF_VALID_START_BEAT flags. The solid vertical lines are measure boundaries, and the dotted lines are beat boundaries. The start point of the segment is aligned with the previous measure boundary in the current primary segment. The segment start time falls at the first beat in the cued segment after the unresolved play time.



Logical Time vs. Actual Time

Some events have both a logical time and an actual time. The actual time is when the event will take place, and the logical time represents the musical position where it belongs.

For example, a segment might contain a program change that belongs to the start of a beat. The logical time is the start of the beat. However, you want to make sure the program change takes place before the note on the beat is played, so you assign it a physical time that's a little earlier.

If the segment loops to the logical time (the start of that same beat), the program change will still go out.

Notification and Event Handling

From time to time, your application may need to respond to a performance event. For example, you might need to know when the end of a segment has been reached, or you might want to synchronize graphics with the beat of the music. You get the desired information by asking DirectMusic to notify you when a certain type of event has taken place.

Note

Performance notifications are not to be confused with DirectSound buffer notifications, which are used only by applications that are streaming wave data directly to or from buffers. See Play Buffer Notification and Capture Buffer Notification.

[C++]
Specify what types of events you want to be notified of by calling the **IDirectMusicPerformance8::AddNotificationType** method once for each desired type of event. The following code example tells DirectMusic to set segment-related events. The actual type of event, such as a segment start or a segment end, will be derived later from the notification message.

```
// pPerformance is a valid IDirectMusicPerformance8 pointer.

GUID guid = GUID_NOTIFICATION_SEGMENT;

pPerformance->AddNotificationType(guid);
```

You can also add notification types for a particular segment by using the **IDirectMusicSegment8::AddNotificationType** method. You could do this, for example, to receive notification of when a particular segment stops playing. You cannot use this method to request GUID_NOTIFICATION_PERFORMANCE types, because these must come from the performance object.

Note

Most applications do not directly call the **IDirectMusicTrack8::AddNotificationType** method.

Information about notifications is sent in **DMUS_NOTIFICATION_PMSG** message structures. You can poll for any pending notification messages within the Windows message loop by calling the **IDirectMusicPerformance8::GetNotificationPMsg** method, or you can have DirectMusic signal an event object in a separate thread when a message is pending.

If you want to be alerted of pending DirectMusic notification messages by a Windows event object, you must first obtain an event handle by calling the Microsoft® Win32® **CreateEvent** function. Typically, you would create an autoreset event with a call such as the following:

```
HANDLE g_hNotify = CreateEvent(NULL, FALSE, FALSE, NULL);
```

After creating the event, assign the handle to the performance by passing it to the **IDirectMusicPerformance8::SetNotificationHandle** method. You can use the second parameter of this method to specify how long DirectMusic should hold onto the event if it is not retrieved. A value of 0 in this parameter indicates that the default time of 2 seconds is to be used.

In the following example, *g_pPerf* is a valid pointer to the **IDirectMusicPerformance8** interface:

```
g_pPerf->SetNotificationHandle(g_hNotify, 0);
```

The following sample function executes repeatedly in its own thread, checking for signaled events and retrieving notification messages:

```
void WaitForEvent( LPVOID lpv)
{
    DWORD dwResult;
    DMUS_NOTIFICATION_PMSG* pPmsg;
    char szCount[4];

    while (TRUE)
    {
        dwResult = WaitForSingleObject(g_hNotify, 100);
        while (S_OK == g_pPerf->GetNotificationPMsg(&pPmsg))
        {
            // Check notification type and do something in response.
            .
            .
            .
            g_pPerf->FreePMsg((DMUS_PMSG*)pPmsg);
        }
    }
}
```

This thread is executed as follows:

```
_beginthread(WaitForEvent, 0, NULL);
```

When notifications are no longer needed, the following code shuts down the thread, removes the notification handle from the performance, and destroys the event object:

```
_endthread();
g_pPerf->SetNotificationHandle(0, 0);
CloseHandle(g_hNotify);
```

It isn't necessary to create an event in order to retrieve notification messages in your application's message loop. As long as you have requested notifications by calling one of the **AddNotificationType** methods, the performance sends messages that can be retrieved by calling **IDirectMusicPerformance8::GetNotificationPMsg**.

More than one message might be waiting when an event is signaled or when you call **GetNotificationPMsg** in the message loop. To be sure of catching all notifications, call **GetNotificationPMsg** repeatedly until it returns S_FALSE.

Multiple messages with the same time stamp are not queued in any particular order.

It is the application's responsibility to free any messages it retrieves, by calling the **IDirectMusicPerformance8::FreePMsg** method.

[Visual Basic]
Specify what types of events you want to be notified of by calling the **DirectMusicPerformance8.AddNotificationType** method once for each desired type of event. The following code example, in which *perf* is a **DirectMusicPerformance8** object, requests notifications for segment events. The type of event (such as a segment start or a segment end) will be derived later from the notification message.

```
Call perf.AddNotificationType(DMUS_NOTIFY_ON_SEGMENT)
```

Notifications are sent in the form of a **DMUS_NOTIFICATION_PMSG** message type. You can poll for any pending notification messages by calling the **DirectMusicPerformance8.GetNotificationPMsg** method in **Sub Main**, or you can have DirectMusic signal an event when a message is pending.

To have DirectMusic signal events, use **DirectX8.CreateEvent** to obtain an event handle, and then pass this handle to **DirectMusicPerformance8.SetNotificationHandle**. The module that you pass to **CreateEvent** must implement the **DirectXEvent8** class and must also provide an implementation of the **DirectXEvent8.DXCallback** method, which is called by DirectMusic whenever an event is signaled.

The following code example sets up notifications for a form module called *frmMain*:

```
' DX is a DirectX8 object; perf is a DirectMusicPerformance8 object.

Dim hEvent As Long
hEvent = DX.CreateEvent(frmMain)
Call perf.SetNotificationHandle(hEvent)
```

The form module contains code to intercept any message indicating that the segment has finished playing:

```
Implements DirectXEvent8

Private Sub DirectXEvent8_DXCallback(ByVal eventid As Long)

  Dim GotMSG As Boolean
  Dim PMsg As DMUS_NOTIFICATION_PMSG

  Do
    GotMSG = gobjDMPerformance.GetNotificationPMSG(PMsg)
    If GotMSG Then
      If PMsg.lNotificationOption = DMUS_NOTIFICATION_SEGEND Then

        ' Segment has finished playing, so do something.
        .
        .
        .
      End If
    End If
  Loop Until Not GotMSG

End Sub
```

It isn't necessary to create an event in order to retrieve notification messages in your application's main loop. As long as you have requested notifications by calling the **DirectMusicPerformance8.AddNotificationType** method, the performance sends messages that can be retrieved by calling **DirectMusicPerformance8.GetNotificationPMsg**.

More than one message might be waiting when an event is signaled or when you call **GetNotificationPMsg** in the loop. To be sure of catching all notifications, call **GetNotificationPMsg** repeatedly until it returns False.

Multiple messages with the same time stamp are not queued in any particular order.

Performance Parameters

[Visual Basic]
In DirectX for Visual Basic, performance parameters are set and retrieved by using various methods of **DirectMusicPerformance8** and **DirectMusicSegment8**.

To have the performance respond immediately to a changed parameter, an application can flush messages from the queue by using the **DirectMusicPerformance8.Invalidate** method. This method causes all tracks to resend messages from the specified point forward.

[C++]
DirectMusic lets you control many aspects of track behavior by changing parameters during playback, using one of the following **SetParam** methods:

- **IDirectMusicPerformance8::SetParam** sets data on a specific track within the current control segment of the performance. The control segment is normally the primary segment, but a secondary segment can be designated as the control segment when it is played. See Control Segments.
- **IDirectMusicSegment8::SetParam** sets data on a specific track within the segment.
- **IDirectMusicTrack8::SetParam** and **IDirectMusicTrack8::SetParamEx** set data on the track represented by the interface. Applications do not normally have interfaces to individual tracks.

The **IDirectMusicPerformance8::SetGlobalParam** method enables you to set values that apply across the entire performance.

The equivalent **GetParam** and **GetGlobalParam** methods retrieve current values for a track or the performance.

To have the music respond immediately to a changed parameter, an application can flush messages from the queue by using the **IDirectMusicPerformance8::Invalidate** method. This method causes all tracks to resend messages from the specified point forward.

More information about parameters is contained in the following topics:

- Setting and Retrieving Track Parameters
- Disabling and Enabling Track Parameters
- Setting and Retrieving Global Parameters

Setting and Retrieving Track Parameters

[Visual Basic]
Some methods of **DirectMusicPerformance8** and **DirectMusicSegment8** have the effect of setting or retrieving parameters on a particular track, usually in the control segment. However, applications using DirectX for Visual Basic don't need to be concerned about tracks as such, and these methods can be used without any knowledge of what is happening at a lower level.

[C++]
The following methods are used for setting and retrieving track parameters:

- **IDirectMusicPerformance8::GetParam**
- **IDirectMusicPerformance8::SetParam**
- **IDirectMusicSegment8::GetParam**
- **IDirectMusicSegment8::SetParam**
- **IDirectMusicTrack8::GetParam**
- **IDirectMusicTrack8::SetParam**
- **IDirectMusicTrack8::GetParamEx**
- **IDirectMusicTrack8::SetParamEx**

When calling one of these methods on the performance or segment, you can identify the track by setting the *dwGroupBits* and *dwIndex* parameters. Usually, however, you can let DirectMusic find the appropriate track for you. For more information, see Identifying the Track.

The track parameter that is being set or retrieved is identified by a GUID in the *rguidType* parameter of the method. Each parameter that requires data is associated with a particular data type, and *pParam* must point to a variable or structure of this type. In some cases, part of the data structure must be initialized even when retrieving the parameter. For some parameters, you must also specify the time within the track at which the change is to take effect or for which the parameter is to be retrieved.

For reference information on the data associated with the standard parameter types, see Standard Track Parameters.

Some parameter changes might not appear to take effect immediately. For example, changing the groove level does not make a difference until the current pattern is about to finish playing and the next pattern is chosen. If you want the change to take effect sooner, you can force the current pattern to be discarded by calling the **IDirectMusicPerformance8::Invalidate** method.

To determine whether a particular parameter is supported by a track, use the **IDirectMusicTrack8::IsParamSupported** method and check for an S_OK result.

More information is given in the following topic:

- Identifying the Track

Identifying the Track

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
When you set or retrieve a parameter by using **IDirectMusicTrack8::SetParamEx** or **IDirectMusicTrack8::GetParamEx**, the parameter is associated with the track on which the method is called. However, when you call **IDirectMusicPerformance8::SetParam**, **IDirectMusicPerformance8::GetParam**, **IDirectMusicSegment8::SetParam**, or **IDirectMusicSegment8::GetParam**, DirectMusic needs to find the appropriate track.

Normally, you can let DirectMusic determine which track contains the desired parameter. To do this, set *dwGroupBits* to 0xFFFFFFFF and *dwIndex* to DMUS_SEG_ANYTRACK or DMUS_SEG_ALLTRACKS. For example, the following call to **IDirectMusicSegment8::SetParam** turns off the tempo track so that looping a segment does not reset the tempo:

```
pIDMSegment->SetParam(GUID_DisableTempo, 0xFFFFFFFF,
    DMUS_SEG_ALLTRACKS, 0, NULL);
```

There are times, however, when you need to identify a track. Typically, this would be the case when a segment contains multiple tracks of the same type. To set or retrieve the parameter on the desired track, you must identify it by group and index value.

Every track belongs to one or more groups, each group being represented by a bit in the *dwGroupBits* parameter of one of the methods under discussion. The track is assigned to a group or groups when it is inserted in the performance. See **IDirectMusicSegment8::InsertTrack**. In the case of segments loaded from a file, track groups are assigned by the author of the segment.

A track is identified by a zero-based index value within each of the groups it belongs to. The index value is determined by the order in which the tracks were inserted.

Suppose a segment contains the tracks shown in the following table.

Track	Group Bits
A	0x1
B	0x2
C	0x1
D	0x3

Group 1 contains tracks A, C, and D, and group 2 contains tracks B and D. If you call the set or get method with a value of 1 in *dwGroupBits* and a value of 0 in *dwIndex*, the parameter is retrieved from track A, which is the first track in group 1. If *dwIndex* is 1, the parameter is retrieved from track C, the second track in the group. Track D belongs to two groups, 1 and 2, so it can be identified as either *dwGroupBits* = 1 and *dwIndex* =2, or *dwGroupBits* = 2 and *dwIndex* = 1.

If you set more than 1 bit in *dwGroupBits*, the parameter is retrieved from the *n*th track containing any of those bits, where *n* is the value in *dwIndex*.

Disabling and Enabling Track Parameters

[C++]
By setting the GUID_DisableTempo and GUID_DisableTimeSig parameters on a track, you can disable the setting of tempo and time signature by a control segment. You might want to do this, for example, when you have set the tempo dynamically and don't want the primary segment to send tempo messages.

To re-enable the parameter, call one of the set-parameter methods with GUID_EnableTempo or GUID_EnableTimeSig as the *rguidType* parameter. You can also set these parameters to force a segment to send tempo messages even though it isn't the control segment, or to cause a secondary segment to send time signature messages. For more information, see Control Segments.

For more information on how to set a parameter, see Setting and Retrieving Track Parameters.

See also the Remarks for **IDirectMusicTrack8::IsParamSupported**.

It is also possible to disable and enable any track parameter by setting the configuration flags on the track. For more information, see Track Configuration.

[Visual Basic]
By using the **DirectMusicSegment8.SetTempoEnable** and **DirectMusicSegment8.SetTimeSigEnable** methods, you can disable the setting of tempo and time signature by a control segment. You might want to do this, for example, when you have set the tempo dynamically and don't want the primary segment to change the tempo.

You can also use **SetTempoEnable** to force a segment to send tempo messages even though it isn't the control segment, and **SetTimeSigEnable** to cause a secondary segment to send time signature messages. For more information, see Control Segments.

Setting and Retrieving Global Parameters

[C++]
By using the **IDirectMusicPerformance8::SetGlobalParam** and **IDirectMusicPerformance8::GetGlobalParam** methods, you can set and retrieve parameters that affect the entire performance rather than a single track.

The parameter to be set or retrieved is identified by a GUID in the *rguidType* parameter of the method. Each parameter is associated with a particular data type, whose size is given in the *dwSize* parameter. The predefined GUIDs and their data types are shown in the following table.

Parameter type GUID (*rguidType*) and Data (pParam*)**	**Description**
GUID_PerfAutoDownload **BOOL**	This parameter controls whether instruments are automatically downloaded when a segment is played. By default, it is off. See Downloading and Unloading Bands.
GUID_PerfMasterGrooveLevel **char**	The master groove level is a value that is always added to the groove level established by the command track. The resulting value is adjusted, if necessary, to fall within the range from 1 through 100.
GUID_PerfMasterTempo **float**	The master tempo is a scaling factor applied to the tempo by the final output tool. By default, it is 1. A value of 0.5 would halve the tempo, and a value of 2.0 would double it. This value can be set in the range from DMUS_MASTERTEMPO_MIN through DMUS_MASTERTEMPO_MAX.
GUID_PerfMasterVolume **long**	The master volume is an amplification or attenuation factor, in hundredths of a decibel, applied to the default volume of the entire performance and any other performances using the same synthesizer. The range of permitted values is determined by the port. For the default software synthesizer, the allowed range is +20db to -200dB, but the useful range is +10db to -100db. Hardware MIDI ports do not support changing master volume.

Applications can also use custom types of global parameters. To create a new type, establish a GUID and a data type for it.

When a parameter is set, the performance allocates memory for the data in a linked list of items identified by GUID. The data can be retrieved by a call to **IDirectMusicPerformance8::GetGlobalParam**.

Note

Even predefined parameters have to be set before they can be retrieved. **GetGlobalParam** fails if **SetGlobalParam** has never been called on the parameter.

[Visual Basic]
The **DirectMusicPerformance8** class has the following methods for setting and retrieving global parameters, which affect the entire performance.

GetMasterAutoDownload **SetMasterAutoDownload**	This parameter controls whether instruments are automatically downloaded when a segment is played. By default, it is off. See Downloading and Unloading Bands.
GetMasterGrooveLevel **SetMasterGrooveLevel**	The master groove level is a value that is always added to the groove level established by the command track. The resulting value is adjusted, if necessary, to fall within the range from 1 through 100.
GetMasterTempo **SetMasterTempo**	The master tempo is a scaling factor that is applied to the tempo by the final output tool. By default, it is 1. A value of 0.5 would halve the tempo, and a value of 2.0 would double it. This value can be set in the range from 0.01 through 100.0.
GetMasterVolume **SetMasterVolume**	The master volume is an amplification or attenuation factor, in hundredths of a decibel, applied to the default volume of the entire performance and any other performances using the same synthesizer. The range of permitted values is determined by the port. Hardware MIDI ports do not support changing master volume.

Using Audiopaths

A DirectMusic performance contains one or more audiopaths, which manage the flow of sound data through various objects. An audiopath might include the performance

itself, a segment, toolgraphs, the synthesizer, DirectSound buffers, effects DMOs, and the primary DirectSound buffer where the final mixing is done.

If your application does nothing more complicated than playing 2-D sound effects or MIDI files, you can set up a standard default audiopath and play everything on it. But to take advantage of the full power of DirectX Audio you must be able to take control of the audiopath.

This section is a guide to creating audiopaths, playing segments on them, and accessing objects within them. The following topics are covered:

- Migrating from Ports to Audiopaths
- Creating Audiopaths
- Default Audiopath
- Standard Audiopaths
- Playing Sounds on Audiopaths
- Retrieving Objects from an Audiopath

Migrating from Ports to Audiopaths

[C++]
If you have written applications using versions of DirectMusic prior to DirectX 8.0, you will find that the recommended way of creating a route for audio output has changed. You can still use DirectMusic the old way, but in order to take advantage of features such as effects processing, individual control of segment states, and custom audiopath configurations in DirectMusic Producer segments, you must do things in a new way.

Important differences between the old and new models for audio output include the following:

- Applications no longer deal directly with ports. DirectMusic handles the creation of the port and the mapping of performance channels. The **IDirectMusicPerformance8::Init**, **IDirectMusicPerformance8::AddPort**, **IDirectMusicPerformance8::AssignPChannel** and **IDirectMusicPerformance8::AssignPChannelBlock** methods are not used in the audiopath model.
- To play sounds through a 3-D buffer, an application no longer needs to create the buffer explicitly and assign it to a port by using **IDirectMusicPort8::SetDirectSound**. The buffer is now created as part of the audiopath, and the application obtains access to it by using one of the **GetObjectInPath** methods.
- Deactivating an audiopath has a different effect than deactivating a port. When an port is deactivated, it no longer produces sound, but the performance can continue playing segments. When an audiopath is deactivated, all playback on that audiopath stops and any attempt to play a segment on it will fail.

[Visual Basic]
If you have written applications using versions of DirectMusic prior to DirectX 8.0, you will find that the way of creating a route for audio output has changed. DirectX for Visual Basic no longer requires you to set a port on the performance. Instead, ports are incorporated into audiopaths.

Creating Audiopaths

[C++]
Applications can set up audiopaths in the following ways:

- Create one or more standard audiopaths by using **IDirectMusicPerformance8::CreateStandardAudioPath**. See Standard Audiopaths.
- Create a default standard audiopath for the performance in the call to **IDirectMusicPerformance8::InitAudio**. See Default Audiopath.
- Obtain an audiopath configuration from a file authored in DirectMusic Producer and pass the configuration object to **IDirectMusicPerformance8::CreateAudioPath**.
- Have DirectMusic create an audiopath from the segment's audiopath configuration when the segment is played. Audiopaths created in this way are temporary and not visible to the application.

An audiopath configuration object can be loaded just like any other object, by using **IDirectMusicLoader8::GetObject** or **IDirectMusicLoader8::LoadObjectFromFile**. A configuration embedded in a segment can be retrieved by using **IDirectMusicSegment8::GetAudioPathConfig**.

The audiopath configuration object does not have a unique interface or methods, and your application cannot change the configuration in any way. All you can do with the object is pass it, by its **IUnknown** interface, to **IDirectMusicPerformance8::CreateAudioPath**. For sample code, see Playing Sounds on Audiopaths.

[Visual Basic]
Applications can set up audiopaths in the following ways:

- Create a default standard audiopath for the performance in the call to **DirectMusicPerformance8.InitAudio**. See Default Audiopath.

- Create one or more standard audiopaths by using **DirectMusicPerformance8.CreateStandardAudioPath**. See Standard Audiopaths.
- Obtain an audiopath configuration from a file authored in DirectMusic Producer and pass the configuration object to **DirectMusicPerformance8.CreateAudioPath**.
- Have DirectMusic create an audiopath from the segment's audiopath configuration when the segment is played. Audiopaths created in this way are temporary and not visible to the application.

An audiopath configuration embedded in a segment can be retrieved by using **DirectMusicSegment8.GetAudioPathConfig**. The audiopath configuration object does not have a unique interface or methods, and your application cannot change the configuration in any way. All you can do with the object is pass it, as a variable of type **IUnknown**, to **DirectMusicPerformance8.CreateAudioPath**. For sample code, see Playing Sounds on Audiopaths.

Audiopath configurations are the only means of creating nonstandard audiopaths. For instance, if different performance channels are to be routed to different buffers, this mapping must be specified in the audiopath configuration of a segment.

Default Audiopath

[C++]
The default audiopath is the one used when a segment is played by using **IDirectMusicPerformance8::PlaySegment**, or when no audiopath is specified in a call to **IDirectMusicPerformance8::PlaySegmentEx**.

You can create an audiopath and make it the default by specifying a standard type in the *dwDefaultPathType* parameter of **IDirectMusicPerformance8::InitAudio**.

Any existing audiopath can be made the default audiopath by passing it to **IDirectMusicPerformance8::SetDefaultAudioPath**. Retrieve the default audiopath by using **IDirectMusicPerformance8::GetDefaultAudioPath**.

[Visual Basic]
The default audiopath is the one used when no audiopath is specified in the call to **DirectMusicPerformance8.PlaySegmentEx**.

You can create an audiopath and make it the default by specifying a standard type in the *lDefaultPathType* parameter of **DirectMusicPerformance8.InitAudio**.

Any existing audiopath can be made the default audiopath by passing it to **DirectMusicPerformance8.SetDefaultAudioPath**. Retrieve the default audiopath by using **DirectMusicPerformance8.GetDefaultAudioPath**.

Standard Audiopaths

If your application does not play exclusively on audiopaths created from audiopath configuration objects, you must create one or more standard audiopaths.

[C++]
Standard audiopaths are identified by the values passed in the *dwType* parameter of **IDirectMusicPerformance8::CreateStandardAudioPath** or in the *dwDefaultPathType* parameter of **IDirectMusicPerformance8::InitAudio**.

[Visual Basic]
Standard audiopaths are identified by the values passed in the *lType* parameter of **DirectMusicPerformance8.CreateStandardAudioPath** or in the *lDefaultPathType* parameter of **DirectMusicPerformance8.InitAudio**.

The audiopaths defined by DirectMusic manage the flow of synthesizer output through combinations of standard buffers, some of which have effect DMOs attached to them. The following table shows the standard audiopaths and which standard buffers they contain. Shared buffers can be used by more than one audiopath.

Audiopath Type	Standard Buffers	Buffer Shared?
DMUS_APATH_DYNAMIC_3D	3-D Dry	No
DMUS_APATH_DYNAMIC_MONO	Mono	No
DMUS_APATH_DYNAMIC_STEREO	Stereo	No
DMUS_APATH_SHARED_STEREOPLUSREVERB	Stereo	Yes
	Reverb	Yes

[C++]
Characteristics of the standard buffers are shown in the following table, where the **Capabilities** column lists values returned in the **dwFlags** member of the **DSBCAPS** structure passed to **IDirectSoundBuffer8::GetCaps**. The last column shows interfaces that can always be obtained from the buffer object. In addition, applications can add effects to buffers by using **IDirectSoundBuffer8::SetFX**, in which case other interfaces will be available. **IDirectSoundBuffer8** is available for all buffers. For more information on obtaining interfaces, see Retrieving Objects from an Audiopath.

Standard Buffer	Description	Capabilities	In
3-D Dry	Mono 3-D buffer	DSBCAPS_CTRL3D DSBCAPS_CTRLFX DSBCAPS_CTRLVOLUME	ID

		DSBCAPS_GLOBALFOCUS DSBCAPS_MUTE3DATMAXDISTANCE	
Mono	Mono buffer with no effects	DSBCAPS_CTRLFX DSBCAPS_CTRLPAN DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS	No
Reverb	Stereo buffer with music reverberation effect	DSBCAPS_CTRLFX DSBCAPS_CTRLPAN DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS	**ID**
Stereo	Stereo buffer with no effects	DSBCAPS_CTRLFX DSBCAPS_CTRLPAN DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS	No

[Visual Basic]
Characteristics of the standard buffers are shown in the following table, where the **Capabilities** column lists values that would be returned in the **lFlags** member of the **DSBCAPS** type passed to **DirectSoundSecondaryBuffer8.GetCaps**. The last column shows objects that can always be obtained from the buffer object. In addition, applications can add effects to buffers by using **DirectSoundSecondaryBuffer8.SetFX**, in which case other objects will be available. A **DirectSoundSecondaryBuffer8** object can be obtained for any buffer. For more information on obtaining objects, see Retrieving Objects from an Audiopath.

Standard Buffer	**Description**	**Capabilities**	**Ob**
3-D Dry	Mono 3-D buffer	DSBCAPS_CTRLFX DSBCAPS_CTRL3D DSBCAPS_CTRLFX DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS DSBCAPS_MUTE3DATMAXDISTANCE	**Di**
Mono	Mono buffer with no effects	DSBCAPS_CTRLFX DSBCAPS_CTRLPAN DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS	No
Reverb	Stereo buffer with music reverberation effect	DSBCAPS_CTRLFX DSBCAPS_CTRLPAN DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS	**Di**
Stereo	Stereo buffer with no effects	DSBCAPS_CTRLFX DSBCAPS_CTRLPAN DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS	No

More information about the standard audiopaths is available in the following topics:

- DMUS_APATH_DYNAMIC_3D
- DMUS_APATH_DYNAMIC_MONO
- DMUS_APATH_DYNAMIC_STEREO
- DMUS_APATH_SHARED_STEREOPLUSREVERB

DMUS_APATH_DYNAMIC_3D

Sets up a nonshared buffer of type 3-D Dry.

[Visual Basic]
Applications can obtain an object representing the 3-D Dry buffer by calling **DirectMusicAudioPath8.GetObjectInPath** with *lStage* set to DMUS_PATH_BUFFER and *lBuffer* set to 0.

[C++]
Applications can obtain an interface to the 3-D Dry buffer by calling one of the **GetObjectInPath** methods with *dwStage* set to DMUS_PATH_BUFFER and *dwBuffer* set to 0.

DMUS_APATH_DYNAMIC_MONO

Sets up a nonshared buffer of type Mono that has no 3-D parameters or special effects.

[Visual Basic]
Applications can obtain an object representing the buffer by calling **DirectMusicAudioPath8.GetObjectInPath** with *lStage* set to DMUS_PATH_BUFFER and *lBuffer* set to 0.

[C++]
Applications can obtain an interface to the buffer by calling one of the **GetObjectInPath** methods with *dwStage* set to DMUS_PATH_BUFFER and *dwBuffer* set to 0.

DMUS_APATH_DYNAMIC_STEREO

Sets up a nonshared buffer of type Stereo. This audiopath is intended for sound effects on stereo buffers. No reverberation is available.

[Visual Basic]
Applications can obtain an object representing the buffer by calling **DirectMusicAudioPath8.GetObjectInPath** with *lStage* set to DMUS_PATH_BUFFER and *lBuffer* set to 0.

[C++]
Applications can obtain an interface to the buffer by calling one of the **GetObjectInPath** methods with *dwStage* set to DMUS_PATH_BUFFER and *dwBuffer* set to 0.

DMUS_APATH_SHARED_STEREOPLUSREVERB

Sets up buffers of type Reverb and Stereo.

The Stereo buffer is shared among multiple audiopaths. It is a sink-in buffer, meaning that it accepts data directly from the synthesizer, not from other buffers.

[Visual Basic]
Applications can obtain an object representing the Stereo buffer by calling **DirectMusicAudioPath8.GetObjectInPath** with *lStage* set to DMUS_PATH_BUFFER and *lBuffer* set to 0.

The Reverb buffer is also a shared sink-in buffer. Unlike the Stereo buffer, it accepts a mono input from the synthesizer and converts the data to stereo format.

Applications can obtain an object representing the Reverb buffer by calling one of the **GetObjectInPath** methods with *dwStage* set to DMUS_PATH_BUFFER and *dwBuffer* set to 1.

[C++]
Applications can obtain an interface to the Stereo buffer by calling one of the **GetObjectInPath** methods with *dwStage* set to DMUS_PATH_BUFFER and *dwBuffer* set to 0.

The Reverb buffer is also a shared sink-in buffer. Unlike the Stereo buffer, it accepts a mono input from the synthesizer and converts the data to stereo format.

Applications can obtain an interface to the Reverb buffer by calling one of the **GetObjectInPath** methods with *dwStage* set to DMUS_PATH_BUFFER and *dwBuffer* set to 1.

The following sample code, where *g_pPerf* is an **IDirectMusicPerformance8**interface, retrieves an **IDirectSoundFXWavesReverb8** interface to the DMO in the Reverb buffer on a default DMUS_APATH_SHARED_STEREOPLUSREVERB audiopath:

```
IDirectMusicAudioPath8 * pAudioPath;
IDirectSoundFXWavesReverb8 * pEffectDMO;
HRESULT hr;

hr = g_pPerf->GetDefaultAudioPath(&pAudioPath);
if (SUCCEEDED(hr))
{
  HRESULT hr = pAudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,
      DMUS_PATH_BUFFER_DMO, 1,
      GUID_All_Objects, 0, IID_IDirectSoundFXWavesReverb8,
      (LPVOID*) &pEffectDMO);
}
```

Playing Sounds on Audiopaths

[C++]
If your application has created a default audiopath in the call to **IDirectMusicPerformance8::InitAudio**, you can play a segment on this path by using **IDirectMusicPerformance8::PlaySegment**. You can also play a segment on the default path by passing NULL in the *pAudioPath* parameter of **IDirectMusicPerformance8::PlaySegmentEx**.

If there is no default audiopath, or if you want to play a segment on another path, you must use **PlaySegmentEx** rather than **PlaySegment**. You can specify the audiopath in two ways:

- Supply a pointer in the *pAudioPath* parameter. Usually this is the **IDirectMusicAudioPath8** interface pointer you received when the audiopath was created.
- Include DMUS_SEGF_USE_AUDIOPATH in *dwFlags*. This flag causes the segment to create an audiopath from a configuration embedded in the segment object.

Note

An audiopath created in response to the DMUS_SEGF_USE_AUDIOPATH flag is automatically released as soon as the segment has stopped playing. If the audiopath contains an effect such as reverberation, the effect is cut short prematurely. To prevent this from happening, the application should create the audiopath manually and release it only after a suitable delay.

Bands are not downloaded for segments played with the DMUS_SEGF_USE_AUDIOPATH flag unless automatic downloading is enabled. For more information, see Automatically Downloading Bands.

The following sample code, where *g_pPerf* is an **IDirectMusicPerformance8** interface pointer and *g_pSeg* is an **IDirectMusicSegment8** interface to any type of sound file, plays the segment on an embedded audiopath configuration if one is available, or on the default audiopath otherwise:

```
IDirectMusicAudioPath8 * pPath =  NULL;
IUnknown *pConfig;

if (g_pSeg)
{
  if (SUCCEEDED(g_pSeg->GetAudioPathConfig(&pConfig)))
  {
    g_pPerf->CreateAudioPath(pConfig, TRUE, &pPath);
    pConfig->Release();
  }
  g_pPerf->PlaySegmentEx(g_pSeg, NULL, NULL, 0, 0, NULL,
      NULL, pPath);
  if (pPath)
  {
      pPath->Release();
      pPath = NULL;
  }
}
```

If you have an interface to the audiopath, you can change the volume by using **IDirectMusicAudioPath8::SetVolume**. Unlike the global parameter GUID_PerfMasterVolume, which affects all sounds playing on the synthesizer, this method sets the volume only on the performance channels playing on this audiopath.

[Visual Basic]
If your application has created a default audiopath in the call to **DirectMusicPerformance8.InitAudio**, you can play a segment on this path by omitting the *AudioPath* parameter of **DirectMusicPerformance8.PlaySegmentEx**.

If there is no default audiopath, or if you want to play a segment on another path, specify the audiopath by using one of the following techniques:

- Supply an **DirectMusicAudioPath8** object in the *AudioPath* parameter. Usually this is the object you received when the audiopath was created.
- Include DMUS_SEGF_USE_AUDIOPATH in *lFlags*. This flag causes the segment to create an audiopath from a configuration embedded in the segment object.

Note

An audiopath created in response to the DMUS_SEGF_USE_AUDIOPATH flag is automatically released as soon as the segment has stopped playing. If the audiopath contains an effect such as reverberation, the effect is cut short prematurely. To prevent this from happening, the application should create the audiopath manually and release it only after a suitable delay.
Bands are not downloaded for segments played with the DMUS_SEGF_USE_AUDIOPATH flag unless automatic downloading is enabled. For more information, see Automatically Downloading Bands.

The following sample code, where *dmp* is a **DirectMusicPerformance8** object and *dmSeg* is a **DirectMusicSegment8** object representing any type of sound file, plays the segment on an embedded audiopath configuration if one is available. If an error occurs on the call to **DirectMusicSegment8.GetAudioPathConfig**, the value of *path* remains Nothing and the segment is played on the default audiopath.

```
Dim path As DirectMusicAudioPath8
Dim Config As IUnknown

If Not (dmSeg Is Nothing) Then

  On Error GoTo NOCONFIG
  Set path = Nothing
  Set Config = dmSeg.GetAudioPathConfig
  Set path = dmp.CreateAudioPath(Config, True)
NOCONFIG:
  On Error Resume Next
  dmp.PlaySegmentEx dmSeg, 0, 0, , path

End If
```

If you have an audiopath object, you can change the volume by using **DirectMusicAudioPath8.SetVolume**. Unlike the **DirectMusicPerformance8.SetMasterVolume** method, which affects all sounds playing on the synthesizer, this method sets the volume only on the performance channels playing on this audiopath.

Retrieving Objects from an Audiopath

[C++]
It is often necessary to retrieve an interface to a particular object within the audiopath. Common reasons to do so include the following:

- To manipulate the 3-D properties of sounds by obtaining an **IDirectSound3DBuffer8** interface.

- To set global 3-D sound properties by obtaining an **IDirectSound3DListener8** interface from the primary buffer.
- To set effects on a secondary buffer by obtaining the **IDirectSoundBuffer8** interface.
- To set effect parameters by obtaining an interface to a DMO such as **IDirectSoundFXI3DL2Reverb8**.

Objects can be retrieved from an audiopath by calling **IDirectMusicSegmentState8::GetObjectInPath** on the segment state that is playing on the audiopath. You can also call **IDirectMusicAudioPath8::GetObjectInPath** on the audiopath object itself. The following table gives information about the parameters to these two methods.

dwStage	***guidObject***	***dwPChannel* used?**	***dwIndex* used?**	***iidInterface* (typical)**
DMUS_PATH_AUDIOPATH (*)	Ignored	No	No	IID_IDirectM udioPath8
DMUS_PATH_AUDIOPATH_GRAPH	Ignored	No	No	IID_IDirectM aph8
DMUS_PATH_AUDIOPATH_TOOL	Tool class ID or GUID_All_Objects to enumerate	Yes	Yes	IID_IDirectM ol8
DMUS_PATH_BUFFER	Ignored	Yes	No, identify by *dwBuffer*	IID_IDirectS uffer8
DMUS_PATH_BUFFER_DMO	DMO class ID, such as GUID_DSFX_STANDARD_GARGLE, or GUID_All_Objects to enumerate	Yes	Yes; index of DMO within buffer	GUID for sta DMO interfa such as IID_IDirectS XGargle(**)
DMUS_PATH_MIXIN_BUFFER	Ignored	No	No, identify by *dwBuffer*	IID_IDirectS uffer8
DMUS_PATH_MIXIN_BUFFER_DMO	DMO class ID, such as GUID_DSFX_STANDARD_I3DL2REVERB, or GUID_All_Objects to enumerate	No	Yes; index of DMO within buffer	GUID for sta DMO interfa such as IID_IDirectS XI3DL2Rev

DMUS_PATH_PERFORMANCE	Ignored	No	No	IID_IDirectM rformance8
DMUS_PATH_PERFORMANCE_GRAPH	Ignored	No	No	IID_IDirectM aph8
DMUS_PATH_PERFORMANCE_TOOL	Tool class ID or GUID_All_Objects to enumerate	Yes	Yes	IID_IDirectM ol8
DMUS_PATH_PORT	Port class ID or GUID_All_Objects to enumerate	Yes	Yes	IID_IDirectM rt
DMUS_PATH_PRIMARY_BUFFER	Ignored	No	No	IID_IDirectS DListener8
DMUS_PATH_SEGMENT (*)	Ignored	No	No	IID_IDirectM gment8
DMUS_PATH_SEGMENT_GRAPH (*)	Ignored	No	No	IID_IDirectM aph8
DMUS_PATH_SEGMENT_TOOL (*)	Tool class ID or GUID_All_Objects to enumerate	Yes	Yes	IID_IDirectM ol8
DMUS_PATH_SEGMENT_TRACK (*)	Track class ID or GUID_All_Objects to enumerate	No	Yes	IID_IDirectM ack8

Notes

(*) Objects in this stage cannot be retrieved by **IDirectMusicAudioPath8::GetObjectInPath**.
(**) The standard DMOs provided with DirectX also support the **IMediaObject**, **IMediaObjectInPlace**, and **IMediaParams** interfaces. For information on these interfaces, see the DirectShow documentation.

For more information on the values for *dwIndex* when retrieving standard buffers, see Standard Audiopaths.

If you already have an interface to an effects buffer, it is also possible to retrieve a DMO interface by using **IDirectSoundBuffer8::GetObjectInPath**.

You can retrieve an **IDirectSoundBuffer8** interface for any buffer in the audiopath, but some methods are not valid. For more information, see **IDirectSoundBuffer8**.

[Visual Basic]
It is often necessary to retrieve a particular object within the audiopath. Common reasons to do so include the following:

- To manipulate the 3-D properties of sounds by obtaining a **DirectSound3DBuffer8** object.
- To set global 3-D sound properties by obtaining an **DirectSound3DListener8** object from the primary buffer.
- To set effects on a secondary buffer by obtaining a **DirectSoundSecondaryBuffer8** object.
- To set effect parameters by obtaining a DMO such as **DirectSoundFXI3DL2Reverb8**.

Objects can be retrieved from an audiopath by calling **DirectMusicAudioPath8.GetObjectInPath**. The following table gives information about the parameters to this method.

lStage	***guidObject***	***lPChannel* used?**	***lIndex* used?**	***iidInterface* (typical)**
DMUS_PATH_AUDIOPATH	Not supported in DirectX for Visual Basic. The application alre has the **DirectMusicAudioPath8** object from which **GetObjectInPath** was called.			
DMUS_PATH_AUDIOPATH_GRAPH	Not supported in DirectX for Visual Basic.			
DMUS_PATH_AUDIOPATH_TOOL	Not supported in DirectX for Visual Basic.			
DMUS_PATH_BUFFER	GUID_ALL	Yes	No, identify by *dwBuffer*.	IID_DirectS SecondaryB
DMUS_PATH_BUFFER_DMO	DMO class ID, such as GUID_DSFX_ STANDARD_ GARGLE, or GUID_ALL to enumerate	Yes	No, identify by *dwBuffer*.	IID for stanc DMO, such as IID_DirectS Gargle
DMUS_PATH_MIXIN_BUFFER	GUID_ALL	No	Yes	IID_DirectS SecondaryB
DMUS_PATH_MIXIN_BUFFER_DMO	DMO class GUID, or GUID_ALL to enumerate	No	Yes	IID for stanc DMO, such as IID_DirectS FXI3DL2Re
DMUS_PATH_PERFORMANCE	Not supported in DirectX for Visual Basic.			

DMUS_PATH_PERFORMANCE_GRAPH	Not supported in DirectX for Visual Basic.			
DMUS_PATH_PERFORMANCE_TOOL	Not supported in DirectX for Visual Basic.			
DMUS_PATH_PORT	Not supported in DirectX for Visual Basic.			
DMUS_PATH_PRIMARY_BUFFER	GUID_ALL	No	No	IID_DirectS 3DListener8
DMUS_PATH_SEGMENT (*)	GUID_ALL	No	No	IID_DirectM Segment8
DMUS_PATH_SEGMENT_GRAPH (*)	Not supported in DirectX for Visual Basic.			
DMUS_PATH_SEGMENT_TOOL (*)	Not supported in DirectX for Visual Basic.			
DMUS_PATH_SEGMENT_TRACK (*)	Not supported in DirectX for Visual Basic.			

For more information on the values for *lIndex* when retrieving standard buffers, see Standard Audiopaths.

If you already have an object for an effects buffer, it is also possible to retrieve a DMO by using **DirectSoundSecondaryBuffer8.GetObjectInPath**.

You can retrieve a **DirectSoundSecondaryBuffer8** object for any buffer in the audiopath, but some methods are not valid. For more information, see **DirectSoundSecondaryBuffer8**.

If you set a global variable to a buffer object obtained from the audiopath, be sure to set the variable to Nothing before calling **DirectMusicPerformance8.CloseDown**. If you fail to do so, **CloseDown** will destroy the buffer and then a memory access violation can occur when the variable goes out of scope as the program exits.

3-D Sound

Using DirectX Audio, you can locate sounds in space and can apply Doppler shift to moving sounds.

3-D effects are applied to individual DirectSound buffers. Because you can direct different sounds along different audiopaths, each with its own buffer, it's easy to apply different parameters to different sounds.

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

- Coordinates of 3-D Space
- Perception of Sound Positions

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

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

Coordinates of 3-D Space

The position, velocity, and orientation of sound sources and listeners in 3-D space are represented by Cartesian coordinates, which are values on three axes: the x-axis, the y-axis, and the z-axis. The axes are relative to a viewpoint established by the application. Values on the x-axis increase from left to right, on the y-axis from down to up, and on the z-axis from near to far.

[C++]
The **D3DVECTOR** structure contains values describing position, velocity, or orientation on the three axes.

[Visual Basic]
The **D3DVECTOR** type contains values describing position, velocity, or orientation on the three axes.

Conventionally, vectors are expressed as three values enclosed in parentheses and separated by commas, in the order (x, y, z).

For position, the values are in meters by default. If your application does not use the meter as its unit of measure for 3-D graphics, you can set a distance factor, which is a floating-point value representing meters per application-specified distance unit. For example, if your application uses feet, it could specify a distance factor of .3048, which is the number of meters in a foot. For more information, see Distance Factor.

For velocity, the vector describes the rate of movement along each axis in units per second. Again, the default unit is meters, but this can be changed by the application.

For orientation, the values are in arbitrary units and are relative to one another. If the base view of the 3-D world is facing north, level to the ground, and the orientation of the listener is (1, 0, 0)—that is, positive on the x-axis and neutral on the other two axes—then the listener is facing due east. If the vector is (1, 0, 1), the listener's orientation is to the northeast. If it is (-1, 0, 1) the listener is facing northwest. Since the values within a vector are not in absolute units, the last example could equally well be expressed as (-5, 0, 5) or (-0.25, 0, 0.25).

You can see how vectors in 2-D space work by drawing them on a sheet of graph paper. Let the values increase from the bottom of the paper to the top and from left to right. A line drawn from (0, 0) to (1, 1) has the same orientation, or direction, as one drawn from (0, 0), to (5, 5). However, the second line indicates a greater distance, or velocity. 3-D vectors work in just the same way, with an additional axis.

Perception of Sound Positions

In the real world, the perception of a sound's position in space is influenced by several factors. Not all of these factors are acoustical; one of the most important is sight. Clues from the sounds themselves include the following:

- *Overall loudness.* As a sound source moves away from the listener, its perceived volume decreases at a fixed rate. This phenomenon is known as rolloff.
- *Interaural intensity difference*. A sound coming from the listener's right sounds louder in the right ear than in the left.
- *Interaural time difference.* A sound emitted by a source to the listener's right will arrive at the right ear slightly before it arrives at the left ear. The duration of this offset is approximately a millisecond.
- *Muffling.* The shape and orientation of the ears ensures that sounds coming from behind the listener are slightly muffled compared with sounds coming from in front. In addition, if a sound is coming from the right, the sound reaching the left ear will be muffled by the mass of the listener's head as well as by the orientation of the left ear.

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

[C++]
In most cases, DirectSound creates virtual 3-D effects on two speakers or a set of headphones. However, it will also take advantage of four-channel systems with WDM drivers if the user has selected the surround sound speaker configuration in Control Panel. For more information, see **DSBUFFERDESC**.

[Visual Basic]
In most cases, DirectSound creates virtual 3-D effects on two speakers or a set of headphones. However, it will also take advantage of four-channel systems with WDM drivers if the user has selected the surround sound speaker configuration in Control Panel. For more information, see **DSBUFFERDESC**.

DirectSound 3-D Buffers

[C++]
Each sound source in a 3-D environment is represented by an **IDirectSound3DBuffer8** interface. This interface is supported only by sound buffers created with the DSBCAPS_CTRL3D flag. Methods of the interface are used to set and retrieve parameters of a single sound source.

[Visual Basic]
Each sound source in a 3-D environment is represented by a **DirectSound3DBuffer8** object. This object controls the 3-D properties of the buffer from which it is obtained. This object is obtainable only from secondary sound buffers created with the DSBCAPS_CTRL3D flag. Methods of the object are used to set and retrieve parameters of a single sound source.

Applications must supply monaural sound sources when using the 3-D capabilities of DirectSound. If you attempt to create a buffer with the DSBCAPS_CTRL3D flag set and a wave format with more than one channel, an error results.

This section describes how your applications can obtain and manage 3-D buffer objects.The following topics are discussed:

- Obtaining the 3-D Buffer Object
- Batch Parameters for 3-D Buffers
- Minimum and Maximum Distances
- Processing Mode
- Buffer Position and Velocity
- Sound Cones

Obtaining the 3-D Buffer Object

[C++]
The **IDirectSound3DBuffer8** interface is obtained from a secondary DirectSound buffer that has been created with the DSBCAPS_CTRL3D flag in the **dwFlags** member of the **DSBUFFERDESC** structure.

If your application is using the DirectMusic performance and audiopaths, you can create an audiopath containing a 3-D buffer by passing DMUS_APATH_DYNAMIC_3D as the *dwType* parameter to **IDirectMusicPerformance8::CreateStandardAudioPath**.

You can also create a suitable audiopath from a configuration object. An audiopath configuration can specify 3-D parameters for a buffer. When the audiopath is created, the 3-D properties of the buffer are initialized with these parameters.

To obtain an interface to a 3-D buffer in an audiopath, call **IDirectMusicAudioPath8::GetObjectInPath** or **IDirectMusicSegmentState8::GetObjectInPath**, with *dwStage* set to DMUS_PATH_BUFFER.

The following sample code creates a standard audio path and retrieves an **IDirectSound3DBuffer8** interface:

```
HRESULT                 hr;
IDirectMusicAudioPath8*  g_p3DAudioPath;
IDirectSound3DBuffer8*   g_pDS3DBuffer;

if (FAILED(hr = pPerformance->CreateStandardAudioPath(
      DMUS_APATH_DYNAMIC_3D, 64, TRUE, &g_p3DAudioPath)))
   return hr;

if (FAILED(hr = g_p3DAudioPath->GetObjectInPath(
      DMUS_PCHANNEL_ALL, DMUS_PATH_BUFFER, 0,
      GUID_NULL, 0, IID_IDirectSound3DBuffer8,
      (LPVOID*) &g_pDS3DBuffer)))
   return hr;
```

If your application is creating and managing its own DirectSound secondary buffers without using the DirectMusic performance, you can retrieve the **IDirectSound3DBuffer8** interface by calling the **IDirectSoundBuffer8::QueryInterface** method on the buffer, as in the following sample code:

```
// Assume that lpDsbSecondary is a valid IDirectSoundBuffer8
// that has been created with DSBCAPS_CTRL3D.

LPDIRECTSOUND3DBUFFER8 lpDs3dBuffer;

HRESULT hr = lpDsbSecondary->QueryInterface(IID_IDirectSound3DBuffer8,
                                (LPVOID *)&lpDs3dBuffer);
```

[Visual Basic]
The **DirectSound3DBuffer8** object is obtained from a secondary DirectSound buffer that has been created with the DSBCAPS_CTRL3D flag in the **lFlags** member of the **DSBUFFERDESC** type.

If your application is using the DirectMusic performance and audiopaths, you can create an audiopath containing a 3-D buffer by passing DMUS_APATH_DYNAMIC_3D as the *lType* parameter to **DirectMusicPerformance8.CreateStandardAudioPath**.

You can also create a suitable audiopath from a configuration object. An audiopath configuration can specify 3-D parameters for a buffer. When the audiopath is created, the 3-D properties of the buffer are initialized with these parameters.

To obtain an object representing a 3-D buffer in an audiopath, call **DirectMusicAudioPath8.GetObjectInPath**, with *lStage* set to DMUS_PATH_BUFFER, as in the following example:

```
Dim dsb As DirectSound3dBuffer8

Set dsb = dmAudioPath.GetObjectinPath(DMUS_PCHANNEL_ALL, _
    DMUS_PATH_BUFFER, 0, GUID_ALL, 0, IID_DirectSound3DBuffer)
```

The following example, where *m_ds* is a **DirectSound8** object, shows how to create a **DirectSound3DBuffer8** object in an application that is not using the DirectMusic performance:

```
Dim dsbd As DSBUFFERDESC
Dim dsBuffer As DirectSoundSecondaryBuffer8
Dim ds3dBuffer As DirectSound3DBuffer8

' Create the secondary buffer.

dsbd.lFlags = DSBCAPS_CTRL3D
Set dsBuffer = m_ds.CreateSoundBufferFromFile("sound.wav", dsbd)

' Obtain the 3-D buffer object.

Set ds3dBuffer = dsBuffer.GetDirectSound3DBuffer
```

Batch Parameters for 3-D Buffers

[C++]
Applications can retrieve or set a 3-D sound buffer's parameters individually or in batches. To set individual values, your application can use the applicable **IDirectSound3DBuffer8** interface method. However, applications often must set or retrieve all the values at once. You can do this with the **IDirectSound3DBuffer8::GetAllParameters** and **IDirectSound3DBuffer8::SetAllParameters** methods.

[Visual Basic]
Applications can retrieve or set a 3-D sound buffer's parameters individually or in batches. To set individual values, your application can use the applicable **DirectSound3DBuffer8** method. However, applications often must set or retrieve all

the values at once. You can do this with the **DirectSound3DBuffer8.GetAllParameters** and **DirectSound3DBuffer8.SetAllParameters** methods.

Parameter changes can also be made more efficiently by flagging them as deferred and then executing them all at once. For more information, see Deferred Settings.

Minimum and Maximum Distances

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

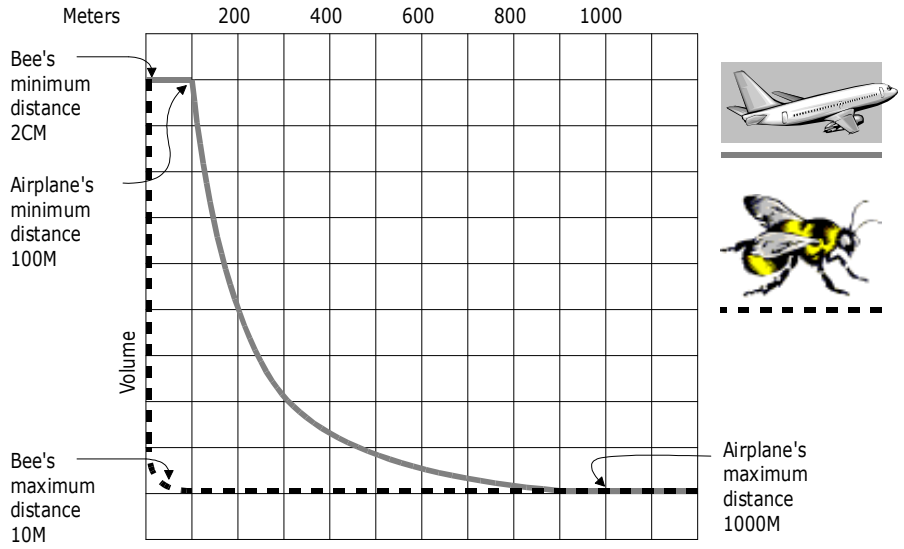
The minimum distance is especially useful when an application must compensate for the difference in absolute volume levels of different sounds. Although a jet engine is much louder than a bee, for practical reasons these sounds must be recorded at similar absolute volumes. An application might use a minimum distance of 100 meters for the jet engine and 2 centimeters for the bee. With these settings, the jet engine would be at half volume when the listener was 200 meters away, but the bee would be at half volume when the listener was 4 centimeters away.

The default minimum distance for a sound buffer, DS3D_DEFAULTMINDISTANCE, is defined as 1 unit, or 1 meter at the default distance factor. Unless you change this value, the sound is at full volume when it is 1 meter away from the listener, half as loud at 2 meters, a quarter as loud at 4 meters, and so on. For most sounds you will probably want to set a larger minimum distance so that the sound does not fade so rapidly as it moves away.

The *maximum distance* for a sound source is the distance beyond which the sound does not get any quieter. The default maximum distance for a DirectSound 3-D buffer (DS3D_DEFAULTMAXDISTANCE) is 1 billion, meaning that in most cases the attenuation will continue to be calculated long after the sound has moved out of hearing range. To avoid unnecessary processing, applications should set a reasonable maximum distance and include the DSBCAPS_MUTE3DATMAXDISTANCE flag when creating the buffer. This flag is automatically set on standard 3-D buffers created as part of an audiopath; see Standard Audiopaths.

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

The following illustration shows how minimum and maximum distance affect the loudness of a jet and a bee at increasing distances.



[C++]
An application sets and retrieves the minimum distance value by using the **IDirectSound3DBuffer8::SetMinDistance** and **IDirectSound3DBuffer8::GetMinDistance** methods. Similarly, it can set and retrieve the maximum distance value by using the **IDirectSound3DBuffer8::SetMaxDistance** and **IDirectSound3DBuffer8::GetMaxDistance** methods.

[Visual Basic]
An application sets and retrieves the minimum distance value by using the **DirectSound3DBuffer8.SetMinDistance** and **DirectSound3DBuffer8.GetMinDistance** methods. Similarly, it can set and retrieve the maximum distance value by using the **DirectSound3DBuffer8.SetMaxDistance** and **DirectSound3DBuffer8.GetMaxDistance** methods.

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

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

Processing Mode

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

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

[C++]
In head-relative mode, the buffer is automatically repositioned in world space as the listener moves and turns. Values set and retrieved through methods such as **IDirectSound3DBuffer8::SetPosition**, **IDirectSound3DBuffer8::SetVelocity**, and **IDirectSound3DBuffer8::GetConeOrientation** are all relative to the current position, velocity, and orientation of the listener.

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

An application sets the mode for a 3-D sound buffer by using the **IDirectSound3DBuffer8::SetMode** method.

[Visual Basic]
In head-relative mode, the buffer is automatically repositioned in world space as the listener moves and turns. Values set and retrieved through methods such as **DirectSound3DBuffer8.SetPosition**, **DirectSound3DBuffer8.SetVelocity**, and **DirectSound3DBuffer8.GetConeOrientation** are all relative to the current position, velocity, and orientation of the listener.

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

An application sets the mode for a 3-D sound buffer by using the **DirectSound3DBuffer8.SetMode** method.

Buffer Position and Velocity

3-D sounds have position and velocity. It is entirely up to the application to specify values for both.

Position is expressed as a vector, relative to either world space or the listener, depending on the processing mode.

Velocity is measured in distance units per second—by default, meters per second. Velocity is used only in calculating the effects of Doppler shift and does not have to correspond to the actual rate of movement of the sound source.

[C++]
An application can set and retrieve a sound source's position in 3-D space by using the **IDirectSound3DBuffer8::SetPosition** and **IDirectSound3DBuffer8::GetPosition** methods.

To set or retrieve the velocity, use the **IDirectSound3DBuffer8::SetVelocity** and **IDirectSound3DBuffer8::GetVelocity** methods.

[Visual Basic]
An application can set and retrieve a sound source's position in 3-D space by using the **DirectSound3DBuffer8.SetPosition** and **DirectSound3DBuffer8.GetPosition** methods.

To set or retrieve the velocity, use the **DirectSound3DBuffer8.SetVelocity** and **DirectSound3DBuffer8.GetVelocity** methods. Velocity is measured in distance units per second—by default, meters per second.

Sound Cones

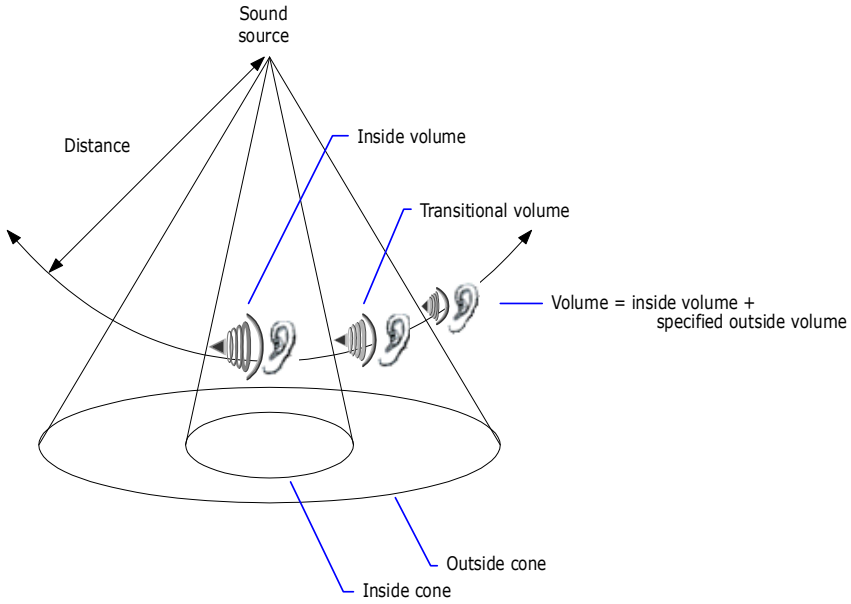
A sound with no orientation has the same amplitude at a given distance in all directions. A sound with an orientation is loudest in the direction of orientation. The model that describes the loudness of the oriented sound is called a sound cone. Sound cones are made up of an inside (or inner) cone and an outside (or outer) cone.

At any angle within the inner cone, the volume of the sound is just what it would be if there were no cone, after taking into account the basic volume of the buffer, the distance from the listener, the listener's orientation, and so on.

At any angle outside the outer cone, the normal volume is attenuated by a factor set by the application. The outside cone volume is expressed in hundredths of decibels and is a negative value, because it represents attenuation from the default volume of 0.

Between the inner and outer cones is a zone of transition from the inside volume to the outside volume. The volume decreases as the angle increases.

The following illustration shows the concept of sound cones.



Every 3-D sound buffer has a sound cone, but by default a buffer behaves like an omnidirectional sound source, because the outside volume is not attenuated, and the inside and outside cone angles are 360 degrees. Unless the application changes these values, the sound does not have any apparent orientation.

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

[C++]
An application sets or retrieves the angles that define sound cones by using the **IDirectSound3DBuffer8::SetConeAngles** and **IDirectSound3DBuffer8::GetConeAngles** methods. The outside cone angle must always be equal to or greater than the inside cone angle.

To set or retrieve the orientation of sound cones, an application can use the **IDirectSound3DBuffer8::SetConeOrientation** and **IDirectSound3DBuffer8::GetConeOrientation** methods.

An application sets and retrieves the outside cone volume by using the **IDirectSound3DBuffer8::SetConeOutsideVolume** and **IDirectSound3DBuffer8::GetConeOutsideVolume** methods.

[Visual Basic]
An application sets or retrieves the angles that define sound cones by using the **DirectSound3DBuffer8.SetConeAngles** and **DirectSound3DBuffer8.GetConeAngles** methods. The outside cone angle must always be equal to or greater than the inside cone angle.

To set or retrieve the orientation of sound cones, an application can use the **DirectSound3DBuffer8.SetConeOrientation** and **DirectSound3DBuffer8.GetConeOrientation** methods.

An application sets and retrieves the outside cone volume by using the **DirectSound3DBuffer8.SetConeOutsideVolume** and **DirectSound3DBuffer8.GetConeOutsideVolume** methods.

DirectSound 3-D Listeners

In a virtual 3-D environment as in the real world, sounds exist only in relation to a point of reception. The 3-D sound effects in a DirectX Audio application are affected not only by the position, orientation, and velocity values of the sound sources, but also by the position, orientation, and velocity of the virtual listener.

By default, the listener is stationary at the zero point of all vectors, oriented with the nose toward the positive Z-axis and the top of the head toward the positive Y-axis. By obtaining an object to represent the listener, the application can change all these values to reflect the movement and facing of the user within virtual space. The listener object also controls the general parameters of the acoustic environment, such as the amount of Doppler shift and the rate of volume attenuation over distance.

This section describes how your application can obtain a listener and manage global 3-D sound parameters. The following topics are discussed:

- Obtaining the 3-D Listener
- Batch Parameters for 3-D Listeners
- Deferred Settings
- Distance Factor
- Listener Orientation
- Listener Position and Velocity
- Doppler Factor
- Rolloff Factor

Obtaining the 3-D Listener

[C++]

Global sound parameters are set and retrieved by using the **IDirectSound3DListener8** interface, which is an interface to the primary sound buffer. There is only one primary buffer in an application, and only one listener.

If you are using the DirectMusic performance and audiopaths to play your 3-D sounds, you can obtain the listener from any audiopath by using **IDirectMusicAudioPath8::GetObjectInPath** or **IDirectMusicSegmentState8::GetObjectInPath**, setting the *dwStage* parameter to DMUS_PATH_PRIMARY_BUFFER. In the following example, *g_p3DAudioPath* is a pointer to an **IDirectMusicAudioPath8** interface:

```
IDirectSound3DListener8* g_pDSListener;

HRESULT hr = g_p3DAudioPath->GetObjectInPath(
    DMUS_PCHANNEL_ALL, DMUS_PATH_PRIMARY_BUFFER, 0,
    GUID_NULL, 0, IID_IDirectSound3DListener8,
    (LPVOID*) &g_pDSListener));
```

If your application is creating and managing its own sound buffers, using only the DirectSound API, you must create a primary buffer object and then obtain the listener interface from that.

Create the primary buffer by using the **IDirectSound8::CreateSoundBuffer** method, specifying the DSBCAPS_CTRL3D and DSBCAPS_PRIMARYBUFFER flags in the **dwFlags** member of the **DSBUFFERDESC** structure. Call the **IDirectSoundBuffer::QueryInterface** method on the resulting buffer to obtain a pointer to an **IDirectSound3DListener8** interface for that buffer.

In the following example, *lpds* is an **IDirectSound8** interface pointer:

```
DSBUFFERDESC        dsbd;
LPDIRECTSOUNDBUFFER    lpdsbPrimary;
LPDIRECTSOUNDLISTENER8 lp3DListener;

ZeroMemory(&dsbd, sizeof(DSBUFFERDESC));
dsbd.dwSize = sizeof(DSBUFFERDESC);
dsbd.dwFlags = DSBCAPS_CTRL3D | DSBCAPS_PRIMARYBUFFER;
if SUCCEEDED(lpds->CreateSoundBuffer(&dsbd, &lpdsbPrimary, NULL))
{
    // Get listener interface.
    if FAILED(lpdsbPrimary->QueryInterface(
          IID_IDirectSound3DListener8,
          (LPVOID *)&lp3DListener))
    {
    lpdsbPrimary->Release();
    }
}
```

Once the listener has been obtained, the **IDirectSoundBuffer** interface to the primary buffer is not needed and can be released.

[Visual Basic]
Global sound parameters are set and retrieved by using the **DirectSound3DListener8** object, which is obtained from the primary sound buffer. There is only one primary buffer in an application, and only one listener.

If you are using the DirectMusic performance and audiopaths to play your 3-D sounds, you can obtain the listener from the audiopath by using **DirectMusicAudioPath8.GetObjectInPath**, setting the *lStage* parameter to DMUS_PATH_PRIMARY_BUFFER. In the following example, *AudioPath* is a **DirectMusicAudioPath8** object:

```
Dim dx As New DirectX8

Dim perf As DirectMusicPerformance8
Dim audparams As DMUS_AUDIOPARAMS
Dim audiopath As DirectMusicAudioPath8
Dim listener As DirectSound3dListener8

Private Sub Form_Load()
  Set perf = dx.DirectMusicPerformanceCreate
  perf.InitAudio Me.hWnd, DMUS_AUDIOF_ALL, audparams, _
     Nothing, Nothing, DMUS_APATH_DYNAMIC_3D, 16
  Set audiopath = perf.GetDefaultAudioPath
  Set listener = audiopath.GetObjectinPath(DMUS_PCHANNEL_ALL, _
     DMUS_PATH_PRIMARY_BUFFER, 0, vbNullString, 0, _
     IID_DirectSound3DListener)
End Sub
```

If your application is creating and managing its own sound buffers using only the DirectSound API, you must create a primary buffer object and then obtain the listener interface from that.

Create the primary buffer object by using the **DirectSound8.CreatePrimarySoundBuffer** method, specifying the DSBCAPS_CTRL3D and DSBCAPS_PRIMARYBUFFER flags in the **lFlags** member of the accompanying **DSBUFFERDESC** type. Call the **DirectSoundPrimaryBuffer8.GetDirectSound3DListener** method on the resulting buffer to obtain a **DirectSound3DListener8** object. These steps are shown in the following example, where *ds* is a global **DirectSound8** object:

```
Dim dsbd As DSBUFFERDESC
Dim dsbPrimary As DirectSoundPrimaryBuffer8
Dim waveFormat As WAVEFORMATEX
Dim ds3dListener As DirectSound3DListener8
```

```
' Create the primary buffer. The waveFormat parameter is ignored
' for primary buffers, so it doesn't have to be initialized.

dsbd.lFlags = DSBCAPS_CTRL3D Or DSBCAPS_PRIMARYBUFFER
Set dsbPrimary = ds.CreatePrimarySoundBuffer(dsbd, waveFormat)

' Create the DirectSound3DListener8 object.

Set ds3DListener = dsbPrimary.GetDirectSound3DListener
```

Once the listener has been obtained, the primary buffer object is not needed and can be allowed to go out of scope.

Batch Parameters for 3-D Listeners

[C++]
Applications can retrieve or set a 3-D listener object's parameters individually or in batches. To set individual values, your application can use the applicable **IDirectSound3DListener8** interface method. However, applications often must set or retrieve all the values that describe the listener at once. An application can perform these batch parameter manipulations in a single call by using the **IDirectSound3DListener8::GetAllParameters** and **IDirectSound3DListener8::SetAllParameters** methods.

[Visual Basic]
Applications can retrieve or set a 3-D listener object's parameters individually or in batches. To set individual values, your application can use the applicable **DirectSound3DListener8** object method. However, applications often must set or retrieve all the values that describe the listener at once. An application can perform these batch parameter manipulations in a single call by using the **DirectSound3DListener8.GetAllParameters** and **DirectSound3DListener8.SetAllParameters** methods.

Parameter changes can also be made more efficiently by flagging them as deferred and then executing them all at once. For more information, see Deferred Settings.

Deferred Settings

[C++]
Every change to 3-D sound buffer and listener settings causes remixing, at the expense of CPU cycles. To minimize the performance impact of changing 3-D

settings, use the DS3D_DEFERRED flag in the *dwApply* parameter of any of the **IDirectSound3DListener8** or **IDirectSound3DBuffer8** methods that change 3-D settings. Then call the **IDirectSound3DListener8::CommitDeferredSettings** method to execute all of the deferred commands at once.

[Visual Basic]
Every change to 3-D sound buffer and listener settings causes remixing, at the expense of CPU cycles. To minimize the performance impact of changing 3-D settings, use the DS3D_DEFERRED flag in the *applyFlag* parameter of any of the **DirectSound3DListener8** or **DirectSound3DBuffer8** methods that change 3-D settings. Then call the **DirectSound3DListener8.CommitDeferredSettings** method to execute all of the deferred commands at once.

Note

Deferred settings are overridden by immediate settings. For example, if you set the listener velocity to (1.0, 0.0, 0.0) with the DS3D_DEFERRED flag and then set it to (2.0, 0.0, 0.0) with the DS3D_IMMEDIATE flag, the velocity becomes (2.0, 0.0, 0.0) and does not change when **CommitDeferredSettings** is called.

Distance Factor

The distance factor is the number of meters in a vector unit. By default, the distance factor is 1.0. If the velocity of a buffer is (2.0, 0.0, 0.0), the sound source is considered to be moving along the x-axis at 2 meters per second. Applications that are using a different unit of measurement for 3-D graphics vectors might want to change the distance factor accordingly.

[C++]
Suppose, for example, that the basic unit of measurement in your application is the foot, or 0.3048 meters. You call the **IDirectSound3DListener8::SetDistanceFactor** method, specifying 0.3048 as the *flDistanceFactor* parameter. From then on, you continue using feet in parameters to method calls, and they are automatically converted to meters.

You can retrieve the current distance factor set for a listener by using the **IDirectSound3DListener8::GetDistanceFactor** method.

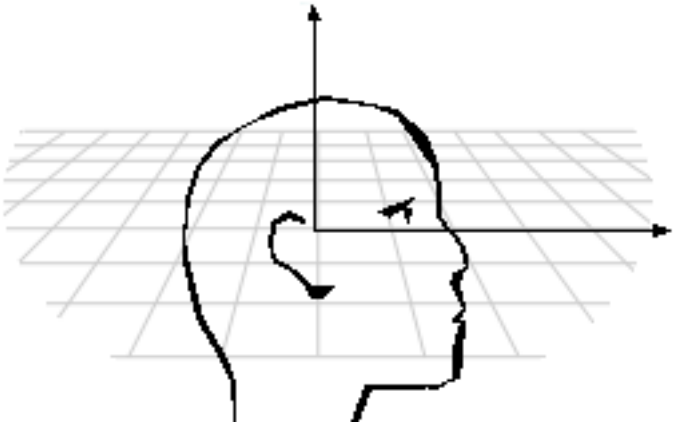
[Visual Basic]
Suppose, for example, that the basic unit of measurement in your application is the foot, or 0.3408 meters. You call the **DirectSound3DListener8.SetDistanceFactor** method, specifying 0.3048 as the *distanceFactor* parameter. From then on, you continue using feet in parameters to method calls, and they are automatically converted to meters.

You can retrieve the current distance factor set for a listener by using the **DirectSound3DListener8.GetDistanceFactor** method.

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

Listener Orientation

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



[C++]
An application can set and retrieve the listener's orientation by using the **IDirectSound3DListener8::SetOrientation** and **IDirectSound3DListener8::GetOrientation** methods.

[Visual Basic]
An application can set and retrieve the listener's orientation by using the **DirectSound3DListener8.SetOrientation** and **DirectSound3DListener8.GetOrientation** methods.

By default, the front vector is (0.0, 0.0, 1.0), and the top vector is (0.0, 1.0, 0.0). The two vectors must always be at right angles to one another. If necessary, DirectSound will adjust the front vector so that it is at right angles to the top vector.

Listener Position and Velocity

[C++]
An application can set and retrieve a listener's position in 3-D space by using the **IDirectSound3DListener8::SetPosition** and **IDirectSound3DListener8::GetPosition** methods.

To set or retrieve the velocity value that DirectSound uses to calculate Doppler-shift effects for a moving listener, use the **IDirectSound3DListener8::SetVelocity** and **IDirectSound3DListener8::GetVelocity** methods. Velocity is measured in distance units per second—by default, meters per second.

[Visual Basic]
An application can set and retrieve a listener's position in 3-D space by using the **DirectSound3DListener8.SetPosition** and **DirectSound3DListener8.GetPosition** methods.

To set or retrieve the velocity value that DirectSound uses to calculate Doppler-shift effects for a moving listener, use the **DirectSound3DListener8.SetVelocity** and **DirectSound3DListener8.GetVelocity** methods. Velocity is measured in distance units per second—by default, meters per second.

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

Doppler Factor

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

In order to have realistic Doppler shift effects in your application, you must calculate the speed of any object that is moving and set the appropriate velocity for that sound source or listener. You are free to exaggerate or lessen this value in a particular case in order to create special effects. You can also globally increase or decrease Doppler shift by changing the Doppler factor.

The Doppler factor can range from DS3D_MINDOPPLERFACTOR to DS3D_MAXDOPPLERFACTOR, defined as 0.0 and 10.0 respectively. A value of 0 means no Doppler shift is applied to a sound. Every other value represents a multiple of the real-world Doppler shift. In other words, a value of 1 (or

DS3D_DEFAULTDOPPLERFACTOR) means the Doppler shift that would be experienced in the real world is applied to the sound, a value of 2 means twice the real-world Doppler shift, and so on.

[C++]
The Doppler factor can be set and retrieved by using the **IDirectSound3DListener8::SetDopplerFactor** and **IDirectSound3DListener8::GetDopplerFactor** methods.

[Visual Basic]
The Doppler factor can be set and retrieved by using the **DirectSound3DListener8.SetDopplerFactor** and **DirectSound3DListener8.GetDopplerFactor** methods.

Rolloff Factor

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

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

[C++]
Set and retrieve the rolloff factor for all sounds by using the **IDirectSound3DListener8::SetRolloffFactor** and **IDirectSound3DListener8::GetRolloffFactor** methods.

[Visual Basic]
Set and retrieve the rolloff factor for all sounds by using the **DirectSound3DListener8.SetRolloffFactor** and **DirectSound3DListener8.GetRolloffFactor** methods.

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

Using Effects

DirectX Audio provides support for effects processing of sounds by DirectX Media Objects (DMOs). For general information on DMOs, see Introduction to DirectX Media Objects in the DirectShow documentation.

A standard set of effects is available to every DirectX application. Other DMOs can be registered on the system.

All the standard DMOs except Waves reverberation can process 8- or 16-bit PCM wave format data with one or two channels at any sample rate supported by DirectSound. Waves reverberation does not support 8-bit samples.

The following topics contain information on setting up and using effects:

- Setting Effects on Buffers
- Effect Parameters
- Standard Effects

This section focuses on the use of effects in audiopaths, but much of the information is also relevant to applications that manage their own DirectSound buffers. See Using Effects in DirectSound.

Setting Effects on Buffers

If you are playing segments authored in DirectMusic Producer with audiopath configurations, any effects are set up when you create the audiopath from the configuration object. A standard audiopath might also contain effects. However, in some cases you may prefer to implement an effect on a custom audiopath at run time or add an effect to a standard audiopath. For example, you might want to add an effect to a standard audiopath so that you can apply the effect to wave or MIDI files.

[C++]
To apply an effect to an audiopath, first obtain an **IDirectSoundBuffer8** interface to a buffer on the path. Then set one or more effects on that buffer by using **IDirectSoundBuffer8::SetFX**.

To learn how to obtain a buffer interface, see Retrieving Objects from an Audiopath. For information on how to identify standard audiopath buffers in the call to **GetObjectInPath**, see the audiopath types under Standard Audiopaths.

The following sample code sets a standard audiopath, retrieves a buffer from the path, and sets an echo effect on the buffer:

```
// Assume that pPerformance is a valid IDirectMusicPerformance8
// interface pointer.

IDirectMusicAudioPath*  g_p3DAudioPath;
IDirectSoundBuffer8*    g_pDSBuffer;
```

```
HRESULT             hr;

// Create a standard audiopath with a source and
// environment reverb buffers. Don't activate the path;
// SetFX fails if the buffer is running.

if( FAILED(hr = pPerformance->CreateStandardAudioPath(
    DMUS_APATH_DYNAMIC_3D, 64, FALSE, &g_p3DAudioPath)))
    return hr;

// Get the buffer in the audio path.

if( FAILED(hr = g_p3DAudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,
    DMUS_PATH_BUFFER, 0, GUID_NULL, 0, IID_IDirectSoundBuffer8,
    (LPVOID*) &g_pDSBuffer)))
    return hr;

// Describe the effect.

DSEFFECTDESC dsEffect;
dsEffect.dwSize = sizeof(DSEFFECTDESC);
dsEffect.dwFlags = 0;
dsEffect.guidDSFXClass = GUID_DSFX_STANDARD_ECHO;
dsEffect.dwReserved1 = 0;
dsEffect.dwReserved2 = 0;

DWORD dwResults;

// Set the effect
if (FAILED(hr = g_pDSBuffer->SetFX(1, &dsEffect, &dwResults)))
    return hr;

// You can check the value of dwResults here to see if and how
// the effect was allocated.

// Activate the path.

g_p3DAudioPath->Activate(TRUE);
```

[Visual Basic]

To apply an effect to an audiopath, first obtain a **DirectSoundSecondaryBuffer8** object for a buffer on the path. Then set one or more effects on that buffer by using **DirectSoundSecondaryBuffer8.SetFX**. The audiopath must be inactive.

To learn how to obtain a buffer object, see Retrieving Objects from an Audiopath. For information on how to identify standard audiopath buffers in the call to **GetObjectInPath**, see the audiopath types under Standard Audiopaths.

The following sample code, where *dmPerformance* is a **DirectMusicPerformance8** object, sets a standard audiopath, retrieves a buffer from the path, and sets an echo effect on the buffer:

```
Dim dmAudioPath As DirectMusicAudioPath8
Dim dsBuffer As DirectSoundSecondaryBuffer8
Dim DSEffects(0) As DSEFFECTDESC
Dim lResults(0) As Long

' Create a standard audiopath with a source and
' environment reverb buffers. Don't activate the path;
' SetFX fails if the buffer is running.

Set dmAudioPath = dmPerformance.CreateStandardAudioPath( _
    DMUS_APATH_DYNAMIC_3D, 128, False)

' Get the buffer in the audio path.

Set dsBuffer = dmAudioPath.GetObjectinPath(DMUS_PCHANNEL_ALL, _
    DMUS_PATH_BUFFER, 0, GUID_ALL, 0, IID_DirectSoundSecondaryBuffer)

' Describe the effect.

DSEffects(0).lFlags = 0
DSEffects(0).guidDSFXClass = DSFX_STANDARD_ECHO

' Set the effect on the buffer.

dsBuffer.SetFX 1, DSEffects, lResults

' You can check the value of lResults(0) here to see if and how
' the effect was allocated.

dmAudioPath.Activate (True)
```

Effect Parameters

[C++]

To set or retrieve parameters of sound effects, you must first obtain the appropriate interface to the buffer that contains the effect. The following interfaces give you access to parameters of the DMOs supplied with DirectX Audio.

- **IDirectSoundFXChorus8**
- **IDirectSoundFXCompressor8**
- **IDirectSoundFXDistortion8**
- **IDirectSoundFXEcho8**
- **IDirectSoundFXFlanger8**
- **IDirectSoundFXGargle8**
- **IDirectSoundFXI3DL2Reverb8**
- **IDirectSoundFXParamEq8**
- **IDirectSoundFXWavesReverb8**

To retrieve an effect interface, pass DMUS_PATH_BUFFER_DMO as the *dwStage* parameter to **IDirectMusicAudioPath8::GetObjectInPath** or **IDirectMusicSegmentState8::GetObjectInPath**. If the audiopath contains more than one buffer, *dwBuffer* must be the index of the buffer. For information on the index numbers of buffers in standard audiopaths, see the audiopath types under Standard Audiopaths.

The following sample code retrieves an interface for the gargle effect on a buffer created from an audiopath configuration. It then uses this interface to change a parameter of the effect.

```
// g_p3DAudioPath is a valid IDirectMusicAudioPath8 pointer
// obtained from IDirectMusicPerformance8::CreateAudioPath.
// To simplify the example, error-handling is omitted.

HRESULT             hr;
IDirectSoundFXGargle8*  pEffectDMO;
DSFXGargle          FXParams;

hr = g_p3DAudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,
      DMUS_PATH_BUFFER_DMO, 0,
      GUID_All_Objects, 0, IID_IDirectSoundFXGargle8,
      (LPVOID*) &pEffectDMO);

hr = pEffectDMO->GetAllParameters(&FXParams);
FXParams.dwRateHz  = DSFXGARGLE_RATEHZ_MIN;
hr = pEffectDMO->SetAllParameters(&FXParams);
```

If your application is managing its own DirectSound buffers, you need to ensure that changes in effect parameters take place immediately. For efficiency, DirectSound processes 100 milliseconds of sound data in a buffer, starting at the play cursor, before **IDirectSoundBuffer8::Play** is called. This can happen after any of the following calls:

- **IDirectSoundBuffer8::Unlock**
- **IDirectSoundBuffer8::Stop**

- **IDirectSoundBuffer8::SetCurrentPosition**
- **IDirectSoundBuffer8::SetFX**

If you call any of these methods and then change effect parameters, the new parameters will not be heard until the preprocessed data has been played. To avoid this situation, do one of the following:

- Write data to the buffer after effect parameters have been changed, not before.
- Call **Stop** or **SetCurrentPosition** on the buffer to force preprocessing after you have set the parameters.

[Visual Basic]

To set or retrieve parameters of sound effects, you must first obtain the appropriate object from the buffer that contains the effect. The following classes give you access to parameters of the DMOs supplied with DirectX Audio:

- **DirectSoundFXChorus8**
- **DirectSoundFXCompressor8**
- **DirectSoundFXDistortion8**
- **DirectSoundFXEcho8**
- **DirectSoundFXFlanger8**
- **DirectSoundFXGargle8**
- **DirectSoundFXI3DL2Reverb8**
- **DirectSoundFXParamEq8**
- **DirectSoundFXWavesReverb8**

To retrieve an effect object, pass DMUS_PATH_BUFFER_DMO as the *lStage* parameter to **DirectMusicAudioPath8.GetObjectInPath**. If the audiopath contains more than one buffer, *lBuffer* must be the index of the buffer. For information on the index numbers of buffers in standard audiopaths, see the audiopath types under Standard Audiopaths.

The following sample code retrieves an interface for the gargle effect on a buffer. It then uses this interface to change a parameter of the effect. Assume that the **DirectMusicAudioPath8** object *dmAudioPath* has been obtained by calling **DirectMusicPerformance8.CreateAudioPath** on an audiopath configuration object.

```
Dim gargleParams As DSFXI3DL2GARGLE
Dim gargleFX As DirectSoundFXGargle8

Set gargleFX = dmAudioPath.GetObjectinPath(DMUS_PCHANNEL_ALL, _
   DMUS_PATH_BUFFER_DMO, 0, DSFX_STANDARD_GARGLE, 0, _
   IID_DirectSoundFXGARGLE)

gargleParams = gargleFX.GetAllParameters
```

```
gargleParams.lRateHZ = DSFXGARGLE_RATEHZ_MIN
gargleFX.SetAllParameters gargleParams
```

If your application is managing its own DirectSound buffers, you might need to take steps to ensure that changes in effect parameters take place immediately. For efficiency, DirectSound processes 100 milliseconds of sound data in a buffer, starting at the play cursor, before **DirectSoundSecondaryBuffer8.Play** is called. This can happen after any of the following calls:

- **DirectSound8.CreateSoundBufferFromFile**
- **DirectSoundSecondaryBuffer8.WriteBuffer**
- **DirectSoundSecondaryBuffer8.Stop**
- **DirectSoundSecondaryBuffer8.SetCurrentPosition**
- **DirectSoundSecondaryBuffer8.SetFX**

If you call any of these methods and then change effect parameters, the new parameters will not be heard until the preprocessed data has been played. To avoid this situation, do one of the following:

- Write data to the buffer after effect parameters have been changed, not before. This technique doesn't apply to buffers created from file, which already contain data.
- Call **Stop** or **SetCurrentPosition** on the buffer to force preprocessing after you have set the parameters.

Standard Effects

This topic is an introduction to the sound effects provided with DirectX Audio. The following effects are covered:

- Chorus
- Compression
- Distortion
- Echo
- Environmental Reverberation
- Flange
- Gargle
- Parametric Equalizer
- Waves Reverberation

Chorus

Chorus is a voice-doubling effect created by echoing the original sound with a slight delay and slightly modulating the delayof the echo.

[Visual Basic]
This effect is represented by the **DirectSoundFXChorus8** object. Its parameters are contained in the **DSFXCHORUS** type.

[C++]
This effect is represented by the **IDirectSoundFXChorus8** interface. Its parameters are contained in the **DSFXChorus** structure.

Compression

Compression is a reduction in the fluctuation of a signal above a certain amplitude.

[Visual Basic]
This effect is represented by the **DirectSoundFXCompressor8** object. Its parameters are contained in the **DSFXCOMPRESSOR** type.

[C++]
This effect is represented by the **IDirectSoundFXCompressor8** interface. Its parameters are contained in the **DSFXCompressor** structure.

Distortion

Distortion is achieved by adding harmonics to the signal in such a way that, as the level increases, the top of the waveform becomes squared off or clipped.

[Visual Basic]
This effect is represented by the **DirectSoundFXDistortion8** object. Its parameters are contained in the **DSFXDISTORTION** type.

[C++]
This effect is represented by the **IDirectSoundFXDistortion8** interface. Its parameters are contained in the **DSFXDistortion** structure.

Echo

An echo effect causes an entire sound to be repeated after a fixed delay.

[Visual Basic]
This effect is represented by the **DirectSoundFXEcho8** object. Its parameters are contained in the **DSFXECHO** type.

[C++]
This effect is represented by the **IDirectSoundFXEcho8** interface. Its parameters are contained in the **DSFXEcho** structure.

Environmental Reverberation

DirectX supports environmental reverberation in accordance with the Interactive 3-D Audio, Level 2 (I3DL2) specification, published by the Interactive Audio Special Interest Group.

The DirectX environmental reverb effect is an implementation of the listener properties in the I3DL2 specification. Source properties are not supported in this release.

Sounds reaching the listener have three temporal components:

- The *direct path* is the audio signal that goes straight from the sound source to the listener, without bouncing off any surface. There is therefore only one direct path signal.
- *Early reflections* are the audio signals that reach the listener after one or two reflections off surfaces such as the walls, floor, and ceiling. If a signal is the result of the sound hitting only one wall on its way to the listener, it is called a first-order reflection. If it bounces off two walls before reaching the listener, it is called a second-order reflection. Humans can typically perceive individual reflections only of the first or second order.
- *Late reverberation*, or simply *reverb*, consists of the combined lower-order reflections, usually a dense succession of echoes of diminishing intensity.

The combination of early reflections and late reverberation is sometimes called the room effect.

Reverb properties include the following:

- Attenuation of the early reflections and late reverberation.
- Rolloff factor, or the rate at which reflected signals become attenuated with distance. The rolloff factor for the direct path is managed by the DirectSound listener.

- Reflections delay. This is the interval between the arrival of the direct-path signals and the arrival of the first early reflections.
- Reverb delay. This is the interval between the first of the early reflections and the onset of late reverberation.
- Decay time. This is the interval between the onset of late reverberation and the time when its intensity has been reduced by 60 dB.
- Diffusion, which is proportional to the number of echos per second in the late reverberation. Depending on the implementation, the density can change as the reverberation decays. In DirectX, the application can control this property by setting a percentage of the range allowed by the implementation.
- Density, which is is proportional to the number of resonances per hertz in the late reverberation. Lower densities produce hollower sounds like those found in small rooms. In DirectX, the application can control this property by setting a percentage of the range allowed by the implementation.

[C++]
Reverb properties are represented by an **IDirectSoundFXI3DL2Reverb8** interface, and the parameters of the environment are contained in a **DSFXI3DL2Reverb** structure.

DirectX supports many sets of default parameters, or presets, that describe the reverb properties of audio environments ranging from mountains to sewer pipes. Most applications can simply choose one of these environments by using **IDirectSoundFXI3DL2Reverb8::SetPreset**.

To set custom properties, use **IDirectSoundFXI3DL2Reverb8::SetAllParameters**. You can retrieve the current properties by using **IDirectSoundFXI3DL2Reverb8::GetAllParameters**.

[Visual Basic]
Reverb properties are represented by a **DirectSoundFXI3DL2Reverb8** object, and the parameters of the environment are contained in a **DSFXI3DL2REVERB** type.

DirectX supports many sets of default parameters, or presets, that describe the reverb properties of audio environments ranging from mountains to sewer pipes. Most applications can simply choose one of these environments by using **DirectSoundFXI3DL2Reverb8.SetPreset**.

To set custom properties, use **DirectSoundFXI3DL2Reverb8.SetAllParameters**. You can retrieve the current properties by using **DirectSoundFXI3DL2Reverb8.GetAllParameters**.

Flange

Flange is an echo effect in which the delay between the original signal and its echo is very short and varies over time. The result is sometimes referred to as a sweeping sound. The term flange originated with the practice of grabbing the flanges of a tape reel to change the speed.

[Visual Basic]
This effect is represented by the **DirectSoundFXFlanger8** object. Its parameters are contained in the **DSFXFLANGER** type.

[C++]
This effect is represented by the **IDirectSoundFXFlanger8** interface. Its parameters are contained in the **DSFXFlanger** structure.

Gargle

The gargle effect modulates the amplitude of the signal.

[Visual Basic]
This effect is represented by the **DirectSoundFXGargle8** object. Its parameters are contained in the **DSFXGARGLE** type.

[C++]
This effect is represented by the **IDirectSoundFXGargle8** interface. Its parameters are contained in the **DSFXGargle** structure.

Parametric Equalizer

A parametric equalizer amplifies or attenuates signals of a given frequency.

Parametric equalizer effects for different pitches can be applied in parallel by setting multiple instances of the GUID_DSFX_STANDARD_PARAMEQ effect on the same buffer. In this way, the application can have tone control similar to that provided by a hardware equalizer.

[Visual Basic]
This effect is represented by the **DirectSoundFXParamEq8** object. Its parameters are contained in the **DSFXPARAMEQ** type.

[C++]

This effect is represented by the **IDirectSoundFXParamEq8** interface. Its parameters are contained in the **DSFXParamEq** structure.

Waves Reverberation

The Waves reverberation effect is intended for use with music. The Waves reverberation DMO is based on the Waves MaxxVerb technology, which is licenced to Microsoft.

[Visual Basic]
This effect is represented by the **DirectSoundFXWavesReverb8** object. Its parameters are contained in the **DSFXWAVESREVERB** type.

[C++]
This effect is represented by the **IDirectSoundFXWavesReverb8** interface. Its parameters are contained in the **DSFXWavesReverb** structure.

Buffer Chains

A sound does not necessarily go through only a single secondary sound buffer. It is possible for buffers in an audiopath to send data to other secondary buffers. The advantage in doing so is that sounds from multiple buffers can be directed to a shared buffer where common 3-D parameters or special effects can be applied. Shared buffers can also be more efficient.

Buffer chains are set up automatically when an audiopath is created from an audiopath configuration embedded in a DirectMusic Producer file. For more information, see Using Audiopaths.

DirectX 8.0 does not support the creation of buffer chains by using the DirectSound API.

Using Compositional Elements

This section is a guide to incorporating dynamic musical components into a DirectX Audio application. It is presumed that you have a basic understanding of elements such as chordmaps and styles. If not, you should first read Compositional Music Elements.

It's possible to incorporate files from DirectMusic Producer into applications without working with individual compositional elements. Many applications use only fully authored segments. However, using individual components gives greater control over the performance at run time.

The following topics are discussed in this section:

- Music Files for Composition
- Overview of Programming for Composition
- Using Styles
- Using Motifs
- Using Chordmaps
- Using Templates
- Using Transitions

Music Files for Composition

[C++]
When programming for DirectMusic composition, you will use a variety of files created in DirectMusic Producer or a similar application. You load these elements into the application as COM objects and obtain interfaces to them. See Loading Audio Data.

The following table summarizes the types of file objects you will encounter. Any of these objects can also be obtained from a container file or from a resource.

The Class GUID is the value that you put in the **guidClass** member of the **DMUS_OBJECTDESC** structure when loading the object.

Element	Class GUID	Interface	File type
Band	CLSID_DirectMusicBand	**IDirectMusicBand8**	bnd
Chordmap	CLSID_DirectMusicChordMap	**IDirectMusicChordMap8**	cdm
DLS collection	CLSID_DirectMusicCollection	**IDirectMusicCollection8**	dls
Segment	CLSID_DirectMusicSegment	**IDirectMusicSegment8**	sgt
Style	CLSID_DirectMusicStyle	**IDirectMusicStyle8**	sty

[Visual Basic]
When programming for DirectMusic composition, you will use a variety of files produced in a tool such as DirectMusic Producer. You load these elements into the application as COM objects and obtain objects for them. See Loading Audio Data.

The following table summarizes the types of file objects you will encounter. Any of these objects can also be obtained from a resource by the corresponding method, such as **DirectMusicLoader8.LoadBandFromResource**.

Element	DirectMusicLoader8 method	Class	File type
Band	**LoadBand**	**DirectMusicBand8**	bnd

Chordmap	**LoadChordMap**	**DirectMusicChordMap8**	cdm
DLS collection	**LoadCollection**	**DirectMusicCollection**	dls
Segment	**LoadSegment**	**DirectMusicSegment8**	sgt
Style	**LoadStyle**	**DirectMusicStyle8**	sty

Note

Bands can be authored as part of a style, in which case they are automatically loaded when the style is loaded. Similarly, styles and bands can be authored into a segment, in which case you don't need separate files for those elements.
Files can also contain references to other files. If a style contains a reference to a band file, the band is automatically loaded when the style is, provided the loader can find the band file.

Overview of Programming for Composition

When you implement music composed at run time, you will use previously authored objects as building blocks. In consultation with the author or other content provider, you can choose to get the musical data in the form of small building blocks that offer you the greatest possible flexibility and variation at run time, or you can use larger prefabricated elements that define the form of the music more fully.

[C++]
Using the largest building blocks, you load highly structured segments based on styles, MIDI files, or waves that contain everything the performance needs in order to play the sound. All you have to do is load the segment and query for the **IDirectMusicSegment8** interface. You pass this interface pointer to the **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx** method. The style playback engine selects pattern variations from the style and plays them according to a fixed chord progression—or, in the case of a MIDI-based segment, simply plays the MIDI sequence. Band changes are usually contained in the segment as well.

[Visual Basic]
Using the largest building blocks, you load highly structured segments (either style-based or MIDI-based) that contain everything the performance needs to know about the music in order to play it. All you have to do is load the segment by using the **DirectMusicLoader8.LoadSegment** or the **DirectMusicLoader8.LoadSegmentFromResource** method. You then pass the returned **DirectMusicSegment8** object to the **DirectMusicPerformance8.PlaySegmentEx** method. The style playback engine selects pattern variations from the style and plays them according to a fixed chord

progression—or, in the case of a MIDI-based segment, simply plays the MIDI sequence. Band changes are usually contained in the segment as well.

If you want to use smaller building blocks, you obtain the following elements:

- Chordmaps, which are road maps of chord progressions.
- Styles, which define a basic melody and rhythm with variations, motifs, and embellishments.
- Template segments, which are structural plans that control various aspects of playback, including the length of the segment, whether it loops, where groove level changes and embellishment patterns are to be placed, and what types of chords in the chordmap are to serve as signposts.

[C++]
You can construct a segment by combining any chordmap, style, and template, using the **IDirectMusicComposer8::ComposeSegmentFromTemplate** method.

To have even more flexibility in music composition at run time, you can create segments based on predefined *shapes* rather than templates, using the **IDirectMusicComposer8::ComposeSegmentFromShape** method. The shape is used in creating the command and signpost tracks, which control the choice of embellishment patterns, the chord progression, and the frequency of chord changes.

When playing segments, you can also control the band used to play the parts. Bands are typically authored right into styles and templates, but they can also be supplied as separate files. To make band changes dynamically, create a secondary segment containing only the band, using the **IDirectMusicBand8::CreateSegment** method, and play this segment when it is time to assign instruments and instrument settings to the primary segment. For more information, see Using Bands.

[Visual Basic]
You can construct a segment by combining any chordmap, style, and template, using the **DirectMusicComposer8.ComposeSegmentFromTemplate** method.

To have even more flexibility in music composition at run time, you can create segments based on predefined *shapes* rather than templates, using the **DirectMusicComposer8.ComposeSegmentFromShape** method. The shape is used in creating the command and signpost tracks, which control the choice of embellishment patterns, the chord progression, and the frequency of chord changes.

When playing segments, you can also control the band used to play the parts. Bands are typically authored right into styles and templates, but they can also be supplied as separate files. To make band changes dynamically, create a secondary segment containing only the band, using the **DirectMusicBand8.CreateSegment** method, and play this segment when it is time to assign instruments and instrument settings to the primary segment. For more information, see Using Bands.

Using Styles

The DirectMusic style object represents a collection of musical patterns, usually including embellishments and motifs, with a time signature, tempo, and band. It defines the basic rhythm and the notes to be played in each instrument part.

For a conceptual overview, see Styles.

A style by itself does not contain enough information to create a segment of music at run time. For this you need two other components: a chordmap, which is a scheme of chord progressions, and a command track to set the groove level and embellishments as the music plays. The command track can come from a template or be generated at run time from a shape. The chordmap generally comes from a chordmap file or resource.

[C++]
To create a segment with a command track based on a template, call the **IDirectMusicComposer8::ComposeSegmentFromTemplate** method. See Using Templates.

To create a segment based on a shape, call the **IDirectMusicComposer8::ComposeSegmentFromShape** method. You supply pointers to a style and a chordmap. You also supply a rate of harmonic motion, which controls the frequency of chord changes, and a shape constant, which determines the progression of groove levels and embellishments.

[Visual Basic]
To create a segment with a command track based on a template, call the **DirectMusicComposer8.ComposeSegmentFromTemplate** method. (See Using Templates.)

To create a segment based on a shape, call the **DirectMusicComposer8.ComposeSegmentFromShape** method. You supply pointers to a style and a chordmap. You also supply a rate of harmonic motion, which controls the frequency of chord changes, and a shape constant, which determines the progression of groove levels and embellishments.

Using Motifs

A motif is a special kind of pattern in a style intended to be played over the basic style pattern, typically in response to an interactive event. Although a motif can be as complex as any other pattern, even containing variations and multiple instrument parts, usually it is a short, simple musical figure that sounds good against a variety of

background patterns. It might also be a sound effect played by a custom DLS instrument or instruments.

[C++]
All the motifs authored into a style become available to you as soon as you have loaded that style. To get a particular motif ready for playback, call the **IDirectMusicStyle8::GetMotif** method, passing in the following parameters:

- The name of the motif. You might know this from the documentation for the style, or you can obtain it from an index value by using the **IDirectMusicStyle8::EnumMotif** method.
- A pointer to receive the **IIDirectMusicSegment8** interface to the segment object to be created by the method.

The following example function obtains and plays the motif whose name is passed in as *pwszMotifName*:

```
void PlayMotif(IDirectMusicPerformance8* pPerf,
       IDirectMusicStyle8* pStyle,
       WCHAR* pwszMotifName)
{
  IDirectMusicSegment* pSeg;

  // Get the motif segment from the style. Check for S_OK
  // specifically, because GetMotif() returns S_FALSE if it
  // does not find the motif.

  if (S_OK == pStyle->GetMotif(pwszMotifName, &pSeg))
  {
    // Play the segment.

    pPerf->PlaySegment(pSeg,
                DMUS_SEGF_BEAT | DMUS_SEGF_SECONDARY,
                0,
                NULL);
    pSeg->Release();
  }
}
```

Note that *pSeg* is played as a secondary segment, because a motif is normally played over a primary segment. You cannot play a motif as a primary segment, because it does not have a chord track or band track. If you do want to play a motif against silence, create a primary segment from a style that has only blank patterns, and keep that segment playing while you play the motif.

[Visual Basic]

All the motifs authored into a style become available to you as soon as you have loaded that style. To get a particular motif ready for playback, call the **DirectMusicStyle8.GetMotif** method, passing in the name of the motif. You might know this from the documentation for the style, or you can obtain it from an index value by using the **DirectMusicStyle8.GetMotifName** method.

The following code example obtains and plays the first motif in the style:

```
' style is a DirectMusicStyle8.
' perf is the DirectMusicPerformance8.

Dim MotifName As String
Dim segMotif As DirectMusicSegment8

MotifName = style.GetMotifName(1)
Set segMotif = style.GetMotif(MotifName)
Call perf.PlaySegmentEx(segMotif, DMUS_SEGF_SECONDARY, 0)
```

Note that *segMotif* is played as a secondary segment, because a motif is normally played over a primary segment. You cannot play a motif as a primary segment, because it does not have a chord track or band track. If you want to play a motif against silence, create a primary segment from a style that has only blank patterns, and keep that segment playing while you play the motif.

Using Chordmaps

A chordmap object represents a collection of chords that provides the foundation of the harmonic structure and the mood of the music. A chordmap contains several pathways with many interconnected chords, providing many possibilities for the composition engine to choose from in determining the chord progression in a piece of music.

For a conceptual overview, see Chordmaps.

For authored segments, applications don't normally need to concern themselves with chordmaps. The chordmap is used at the authoring stage to create a fixed chord progression. However, chordmaps can be used to compose segments at run time and to alter the chord progression of existing segments.

[C++]
If a chordmap reference has been authored into a style, you can retrieve a pointer to its **IDirectMusicChordMap8** interface by passing its name (assigned by the author) to the **IDirectMusicStyle8::GetChordMap** method. You can also use the **IDirectMusicStyle8::EnumChordMap** method to search for a particular chordmap, or the **IDirectMusicStyle8::GetDefaultChordMap** method to obtain a pointer to the default chordmap for the style.

Note
DirectMusic Producer does not support authoring chordmap references into style files.

You set the chordmap for a composition when you create a segment by using either **IDirectMusicComposer8::ComposeSegmentFromTemplate** or **IDirectMusicComposer8::ComposeSegmentFromShape**. For more information, see Using Styles.

Once a segment has been created, you can change its chordmap by calling the **IDirectMusicComposer8::ChangeChordMap** method. This has the effect of changing the mood of the music without altering its basic rhythm and melody.

Every chordmap has an underlying scale, consisting of 24 tones. You can determine the tones of the scale by using the **IDirectMusicChordMap8::GetScale** method. The lower 24 bits of the variable pointed to by the *pdwScale* parameter of this method are set or cleared depending on whether the corresponding tone is part of the scale. The upper 8 bits give the root of the scale as an integer in the range from 0 through 23 (low C to middle B).

[Visual Basic]
You set the chordmap for a composition when you create a segment by using either **DirectMusicComposer8.ComposeSegmentFromTemplate** or **DirectMusicComposer8.ComposeSegmentFromShape**. For more information, see Using Styles.

Once a segment has been created, you can change its chordmap by calling the **DirectMusicComposer8.ChangeChordMap** method. This has the effect of changing the mood of the music without altering its basic rhythm and melody.

Using Templates

A template is a segment that can be used in composing a playable segment of music at run time. The template sets the length of the segment and any loop points. It can provide the command track, which controls changes in the groove level and the choice of embellishment patterns. It also prescribes how the chordmap is used in composing the segment, by specifying from which signpost group each new chord must come.

For a conceptual overview, see Templates.

[C++]
A template is represented by a DirectMusic segment object.

There are two ways to obtain a template:

- Load it from a segment file or resource. You load it as a DirectMusicObject and query for the **IDirectMusicSegment8** interface. For more information, see Loading Audio Data.
- Create it from a shape, using the **IDirectMusicComposer8::ComposeTemplateFromShape** method. You choose the length, the overall shape, whether intro and end embellishment patterns are to be played, and how long the ending is to be. You get back a pointer to the **IDirectMusicSegment** interface.

Once you have obtained a template segment object, you can pass it to the **IDirectMusicComposer8::ComposeSegmentFromTemplate** method, along with pointers to a style and a chordmap. You also supply a rate of harmonic motion, which sets the frequency of chord changes. The **ComposeSegmentFromTemplate** method creates a segment and returns a pointer to its **IDirectMusicSegment8** interface. You pass this pointer to the **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx** method.

[Visual Basic]
A template is represented by a **DirectMusicSegment8** object.

There are two ways to obtain a template object:

- Load it from a template file or resource by using **DirectMusicLoader8.LoadSegment** or **DirectMusicLoader8.LoadSegmentFromResource**.
- Create it from a shape, using the **DirectMusicComposer8.ComposeTemplateFromShape** method. You choose the length, the overall shape, whether intro and end embellishment patterns are to be played, and how long the ending is to be.

Once you have obtained a template segment object, you can pass it to the **DirectMusicComposer8.ComposeSegmentFromTemplate** method, along with pointers to a style and a chordmap. You also supply a rate of harmonic motion, which sets the frequency of chord changes. The **ComposeSegmentFromTemplate** method creates a segment that can be passed to the **DirectMusicPerformance8.PlaySegmentEx** method.

Using Transitions

In order to avoid a sudden and perhaps discordant break when stopping one segment and beginning another, or when bringing the music to a close, you can have the composer object create an intermediate or closing segment that provides an appropriate transition.

[C++]
You have your choice of three techniques for composing transitional segments:

- The **IDirectMusicPerformance8::PlaySegmentEx** method allows you to specify a segment in the *pTransition* parameter. This segment is used as a template for a newly composed transition. The transition is played at *i64StartTime*, and then the segment specified by *pSource* is played.
- The **IDirectMusicComposer8::AutoTransition** method, given a pointer to the performance, creates a transition from the currently playing segment to a second segment of your choice, and then automatically cues the transitional segment and the second segment for playback, returning an **IDirectMusicSegmentState** interface for both. The transition begins playing immediately or on the next boundary, as specified in the *dwFlags* parameter. Optionally, the second segment can be NULL so that the transition is to silence.
- The **IDirectMusicComposer8::ComposeTransition** method composes a transition from any point in one segment to the beginning of a second segment, or to silence, and returns an **IDirectMusicSegment** interface so that the application can play the transition.

The **AutoTransition** and **ComposeTransition** methods both take a chordmap, a command, and a set of flags as parameters:

- The chordmap, as usual, is used to create a chord track that defines the chord progression in the segment.
- The command is one of the **DMUS_COMMANDT_TYPES** enumeration. It determines which type of pattern—either an ordinary groove pattern or one of the embellishments—is called for in the command track of the transitional segment. When the segment plays, an appropriate pattern is selected from the style.
- The flags are from **DMUS_COMPOSEF_FLAGS** and further define the transition, principally its timing. The DMUS_COMPOSEF_MODULATE flag can be used to cause the transition to move smoothly from one tonality to another; it can't be used when there is no second segment, because there can be no modulation to silence.

[Visual Basic]
You have your choice of three techniques for composing transitional segments:

- The **DirectMusicPerformance8.PlaySegmentEx** method allows you to specify a segment in the *Transition* parameter. This segment is used as a template for a newly composed transition. The transition is played at *startTime*, and then the segment specified by *Source* is played.
- The **DirectMusicComposer8.AutoTransition** method, given a **DirectMusicPerformance8** object, creates a transition from the currently playing segment to a second segment of your choice, and then automatically cues

the transitional segment and the second segment for playback. The transition begins playing immediately or on the next boundary, as specified in the *lFlags* parameter. Optionally, the second segment can be Nothing so that the transition is to silence.

- The **DirectMusicComposer8.ComposeTransition** method composes a transition from any point in one segment to the beginning of a second segment, or to silence, and returns a **DirectMusicSegment8** object so that the application can play the transition.

The **AutoTransition** and **ComposeTransition** methods both take a chordmap, a command, and a set of flags as parameters.

- The chordmap, as usual, is used to create a chord track that defines the chord progression in the segment.
- The command is one of the **CONST_DMUS_COMMANDT_TYPES** enumeration. It determines which type of pattern—either an ordinary groove pattern or one of the embellishments—is called for in the command track of the transitional segment. When the segment plays, an appropriate pattern is selected from the style.
- The flags are from **CONST_DMUS_COMPOSEF_FLAGS** and further define the transition, principally its timing. The DMUS_COMPOSEF_MODULATE flag can be used to cause the transition to move smoothly from one tonality to another; it can't be used when there is no second segment, because there can be no modulation to silence.

Transitions created by **AutoTransition** and **ComposeTransition** are normally a single measure in length. However, they can be longer if the DMUS_COMPOSEF_LONG flag is included and the embellishment in the style is more than one measure long. They also contain at least two measures if they are of type DMUS_COMMANDT_ENDANDINTRO.

Track Composition

[C++]
Most tracks in a segment are fixed and generate the same data each time the segment is played. However, tracks can generate their data dynamically each time they start playing or each time they loop, provided they implement the **IDirectMusicTrack8::Compose** method and are configured to do so.

One standard track, the signpost track, supports composition. The signpost track composes a new chord track from a chordmap.

Normally the track configuration is set by the author, but the application can turn track composition behaviors on and off by passing one or more of the following flags

to **IDirectMusicSegment8::SetTrackConfig** or **IDirectMusicSegmentState8::SetTrackConfig**.

DMUS_TRACKCONFIG_COMPOSING
The track is composed by the **IDirectMusicSegment8::Compose** method.

DMUS_TRACKCONFIG_LOOP_COMPOSE
The track is automatically composed each time the segment loops.

DMUS_TRACKCONFIG_PLAY_COMPOSE
The track is automatically composed each time the segment starts.

Automatic composition can take place only when the segment contains a track in which to put the composed content. When the signpost track is composed, it requires a chord track for the new chords. You can ensure that the necessary tracks exist by calling **IDirectMusicSegment8::Compose** before playing the segment.

If you choose to do all composition manually, the only configuration flag to set is DMUS_TRACKCONFIG_COMPOSING. Provided neither of the other two flags is set, the tracks will be composed only when you call **Compose**.

[Visual Basic]
Most tracks in a segment are fixed and generate the same data each time the segment is played. However, tracks can generate their data dynamically each time they start playing or each time they loop, provided they are configured to do so.

One standard track, the signpost track, supports composition. The signpost track composes a new chord track from a chordmap.

Using Audio Scripts

Scripts are objects authored in an application that contains a script editor, such as DirectMusic Producer. Although they consist mainly of text, script objects also contain a few binary parameters.

[C++]
For a conceptual overview, see Audio Scripts.

Load a script by using the **IDirectMusicLoader8::GetObject** method. Obtain the **IDirectMusicScript8** interface, then call **IDirectMusicScript8::Init** to associate the script with a performance.

The following code loads and initializes a script. Assume that *m_pLoader* is an **IDirectMusicLoader8** interface, *m_PPerformance* is an **IDirectMusicPerformance8** interface, and *wstrFileName* contains the name of the file.

```
IDirectMusicScript8*  pScript = NULL;
```

```
DMUS_SCRIPT_ERRORINFO errInfo;
HRESULT             hr;

if (SUCCEEDED(hr = m_pLoader->LoadObjectFromFile(
      CLSID_DirectMusicScript, IID_IDirectMusicScript8,
      wstrFileName, (LPVOID*) &pScript)))
{
   hr = pScript->Init( m_pPerformance, &errInfo);
}
```

Apart from **Init**, the methods of **IDirectMusicScript8** have three main purposes:

- Set and retrieve the value of variables declared in the script. Because script routines do not accept parameters, variables are the only way for the script and the application to exchange information.
- Call routines. A routine must finish executing before the application thread can continue.
- Enumerate routines and variables. These methods are of interest chiefly to script-editing applications.

All the methods of **IDirectMusicScript8**, except the enumeration methods, retrieve error information in a **DMUS_SCRIPT_ERRORINFO** structure. An error can occur if a variable is not found or code within a routine fails to execute.

Scripts can also be used without being directly loaded or called by the application. A segment authored in DirectMusic Producer can contain a script track that triggers calls to routines in one or more scripts.

[Visual Basic]
DirectX for Visual Basic does not enable applications to load and call audio scripts through the API. However, a segment authored in DirectMusic Producer can contain a script track that triggers calls to routines in one or more scripts.

Wave Playback in DirectSound

This section is a guide to playing waves through DirectSound buffers without making use of the services provided by the DirectMusic loader and performance objects.

For most applications written for DirectX 8.0 and later, the best way to play wave sounds is to load them as segments and play them through the performance. Doing so requires less code, allows tighter integration of sound effects and music, and provides greater functionality such as the ability to mix sounds on an audiopath before they enter a 3-D buffer. However, you can still play wave sounds directly to buffers if you so choose.

Information in this section is contained in the following topics:

- DirectSound Playback Overview
- DirectSound Devices
- DirectSound Buffers
- Using Wave Data
- Using Effects in DirectSound

DirectSound Playback Overview

The DirectSound object represents a device and is used to manage that device and create sound buffers.

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

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

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

A short sound can be loaded into a buffer in its entirety and played at any time by a simple method call. Longer sounds have to be streamed. An application can ascertain when it is time to stream more data into the buffer, either by polling the position of the play cursor or by requesting notification when the play cursor reaches certain points.

[C++]
It is your responsibility to put data in the correct format into the secondary sound buffers. DirectSound does not include methods for parsing a wave file or resource. However, there is code in the SDK sample applications that helps with this task.

[Visual Basic]
DirectX for Visual Basic has methods for creating sound buffers that automatically load a complete sound from a wave file or resource. For longer sounds, the application must handle the parsing and streaming.

DirectSound Devices

The first step in implementing DirectSound in an application is to create a DirectSound object, which represents a sound device.

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

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

Enumeration of Sound Devices

[C++]
For an application that is simply going to play sounds through the user's preferred playback device, you don't need to enumerate the available devices. When you create the DirectSound object by calling the **DirectSoundCreate8** function, you can specify a default device. For more information, see Creating the DirectSound Object.

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

Enumeration serves three purposes:

- Reports what hardware is available.
- Supplies a globally unique identifier (GUID) for each device.
- Enables you to create a temporary DirectSound object for each device as it is enumerated, so that you can check the capabilities of the device.

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

The following callback function adds information about each enumerated device to a combo box. Values for the first three parameters are supplied by the device driver. The fourth parameter is passed on from the **DirectSoundEnumerate** function.

```
BOOL CALLBACK DSEnumProc(LPGUID lpGUID,
```

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

   if ( lpGUID != NULL )
   {
      if (( lpTemp = malloc( sizeof(GUID))) == NULL )
      return( TRUE );

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

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

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

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

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

Note

The first device enumerated is always called the Primary Sound Driver, and the *lpGUID* parameter of the callback is NULL. This device represents the preferred playback device set by the user in Control Panel. It is enumerated separately to make it easy for the application to add "Primary Sound Driver" to a list when presenting the user with a choice of devices. The primary device is also enumerated with its proper name and GUID.

[Visual Basic]

For an application that is simply going to play sounds through the user's preferred playback device, you don't need to enumerate the available devices. When you create the **DirectSound8** object by calling the **DirectX8.DirectSoundCreate** function, you can specify a default device. For more information, see Creating the DirectSound Object.

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

Enumeration serves three purposes:

- Reports what hardware is available.
- Supplies a globally unique identifier (GUID) for each device.
- Enables you to create a temporary DirectSound object for each device as it is enumerated, so that you can check the capabilities of the device.

To enumerate devices, you must first call the **DirectX8.GetDSEnum** method. This method creates a **DirectSoundEnum8** object containing a collection of DirectSound-capable devices. Call the **DirectSoundEnum8.GetCount** method to ascertain the number of available devices. Information on each device is returned by the **DirectSoundEnum8.GetDescription**, **DirectSoundEnum8.GetGuid**, and **DirectSoundEnum8.GetName** methods, each of which takes as a parameter the index of a device within the collection.

The following sample application prints the description of each sound device:

```
Dim dx As New DirectX8
Dim dsenum As DirectSoundEnum8

Private Sub Form_Load()
  Set dsenum = dx.GetDSEnum
  For x = 1 To dsenum.GetCount
    Debug.Print dsenum.GetDescription(x), " ",
  Next x
End Sub
```

Note

The first device enumerated is always called "Primary Sound Driver", and its GUID is all zeros. This device represents the preferred playback device set by the user in Control Panel. It is enumerated separately to make it easy for the application to add "Primary Sound Driver" to a list when presenting the user with a choice of devices. The primary device is also enumerated with its proper name and GUID.

Creating the DirectSound Object

[C++]
The simplest way to create the DirectSound object is with the **DirectSoundCreate8** function. The first parameter of this function specifies the GUID of the device to be associated with the object. You can obtain this GUID by Enumeration of Sound Devices, or you can pass one of the following GUIDs to specify a default device:

DSDEVID_DefaultPlayback
: The default system audio device. You can also specify this device by passing a NULL pointer in the device GUID parameter. The default device is the one enumerated as "Primary DirectSound Driver".

DSDEVID_DefaultVoicePlayback
: The default voice communications device. Typically this is a secondary device such as a USB headset with microphone.

If no device driver is present, the call to **DirectSoundCreate8** fails.

The function returns an error if there is no sound device or, under VXD drivers, if the sound device is under the control of an application using the standard Win32 waveform-audio functions. You should prepare your applications for this call to fail so that they can either continue without sound or prompt the user to close the application that is already using the sound device.

The following code creates a DirectSound object for the default device and obtains the **IDirectSound8** interface:

```
LPDIRECTSOUND8 lpds;
HRESULT hr = DirectSoundCreate8(NULL, &lpds, NULL));
```

Note

DirectSoundCreate8 does not cause **CoInitialize** to be called. If your application uses effect DMOs, it must call **CoInitialize** regardless of how the DirectSound object is created.

If your application will capture sounds as well as play them, you can conveniently create both the DirectSound and the DirectSoundCapture object, as well as playback and capture buffers, by using the **DirectSoundFullDuplexCreate8** function.

You can also create the DirectSound object by using standard COM functions, as follows:

1. Initialize COM at the start of your application by calling **CoInitializeEx**:

   ```
   if FAILED(CoInitializeEx(NULL, 0))
     return FALSE;
   ```

2. Create the DirectSound object by using **CoCreateInstance** and the **IDirectSound8::Initialize** method, rather than the **DirectSoundCreate8** function:

   ```
   LPDIRECTSOUND8 lpds;
   ```

```
HRESULT hr = CoCreateInstance(&CLSID_DirectSound8,
                              NULL,
                              CLSCTX_INPROC_SERVER,
                              &IID_IDirectSound8,
                              &lpds);
```

CLSID_DirectSound8 is the class identifier of the DirectSound driver object class and *IID_IDirectSound8* is the interface identifier. The *lpds* parameter receives the interface pointer.

3. Call the **IDirectSound8::Initialize** method to associate the object with a device. This method takes the same device GUID parameter that **DirectSoundCreate8** uses.

```
if SUCCEEDED(hr)
  hr = lpds->Initialize(NULL);
```

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

```
CoUninitialize();
```

[Visual Basic]

Create the **DirectSound8** object by using the **DirectX8.DirectSoundCreate** method. The single parameter specifies the GUID of the device to be associated with the object. You can obtain this GUID by Enumeration of Sound Devices, or you can specify a default device by passing one of the following values:

DSDEVID_DEFAULTPLAYBACK
: The default system audio device. You can also specify this device by passing **vbNullString** in the device GUID parameter. The default device is the one enumerated as "Primary DirectSound Driver".

DSDEVID_DEFAULTVOICEPLAYBACK
: The default voice communications device. Typically this is a secondary device such as a USB headset with microphone.

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

The following example code creates a **DirectSound8** object using the default system device:

```
Dim m_dx As New DirectX8
Dim m_ds As DirectSound8
Set m_ds = m_dx.DirectSoundCreate(vbNullString)
```

Cooperative Levels

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

[C++]
After creating a DirectSound object, you must set the cooperative level for the device by using the **IDirectSound8::SetCooperativeLevel** method before you can play sounds.

The following example sets the cooperative level for the DirectSound device represented by the **IDirectSound8** interface at *lpDirectSound*. The *hwnd* parameter is the handle to the application window.

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

[Visual Basic]
After creating a **DirectSound8** object, you must set the cooperative level for the device with the **DirectSound8.SetCooperativeLevel** method before you can play sounds.

The following Visual Basic example sets the cooperative level for the **DirectSound8** object *m_ds*.. The *hwnd* argument is the handle to the application window.

```
m_ds.SetCooperativeLevel Me.hWnd, DSSCL_PRIORITY
```

DirectSound defines three cooperative levels for sound devices, represented by the values DSSCL_NORMAL, DSSCL_PRIORITY, and DSSCL_WRITEPRIMARY.

Note

The DSSCL_EXCLUSIVE cooperative level available in versions earlier than DirectX 8.0 is obsolete. It is no longer possible for a DirectX application to mute other applications. Applications that request the exclusive level are granted the priority level instead.

Normal Cooperative Level

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

Priority Cooperative Level

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

Game applications should use the priority cooperative level in almost all circumstances. This level gives the most robust behavior while allowing the application control over sampling rate and bit depth. The priority cooperative level also allows audio from other applications, such as IP telephony, to be heard along with the audio from the game.

Write-primary Cooperative Level

[Visual Basic]
Visual Basic applications should not use the DSSCL_WRITEPRIMARY cooperative level. This level is for specialized applications that do not use the DirectSound mixer.

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

An application must be set to the write-primary level in order to obtain direct write access to the audio samples in the primary buffer. If the application is not set to this level, then all calls to the **IDirectSoundBuffer8::Lock** method will fail.

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

You cannot set the write-primary cooperative level if a DirectSound driver is not present on the user's system. To determine whether this is the case, call the **IDirectSound8::GetCaps** method and check for the DSCAPS_EMULDRIVER flag in the **DSCAPS** structure.

For more information, see Writing to the Primary Buffer.

Device Capabilities

DirectSound enables your application to retrieve the hardware capabilities of the sound device. Most applications will not need to do this, because DirectSound automatically takes advantage of any available hardware acceleration. However, high-performance applications can use the information to scale their sound requirements to

the available hardware. For example, an application might choose to play more sounds if hardware mixing is available than if it is not.

[C++]
After calling the **DirectSoundCreate8** function to create a DirectSound object, your application can retrieve the capabilities of the sound device by calling the **IDirectSound8::GetCaps** method.

The following example retrieves the capabilities of the device represented by the **IDirectSound8** interface pointer *lpDirectSound*:

```
DSCAPS dscaps;

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

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

If your application scales to hardware capabilities, you should call the **IDirectSound8::GetCaps** method between every buffer allocation to determine if there are enough resources to create the next buffer.

[Visual Basic]
After calling the **DirectX8.DirectSoundCreate** method to create a **DirectSound8** object, your application can retrieve the capabilities of the sound device by calling the **DirectSound8.GetCaps** method.

The following example retrieves the capabilities of the device represented by the **DirectSound8** object *m_ds*:

```
Dim caps As DSCAPS
m_ds.GetCaps caps
```

The **DSCAPS** type receives information about the performance and resources of the sound device, including the maximum resources of each type and the resources that are currently available.

If your application scales to hardware capabilities, you should call the **DirectSound8.GetCaps** method between every buffer allocation to determine if there are enough resources to create the next buffer.

Speaker Configuration

DirectSound uses the speaker configuration—that is, the position of the speakers relative to the listener—to optimize 3-D effects for the user's sound system.

[C++]
In Windows® 98 and Windows® 2000, the speaker configuration can be set by the user in Control Panel. An application can retrieve this value by using **IDirectSound8::GetSpeakerConfig** and override it by calling **IDirectSound8::SetSpeakerConfig**.

[Visual Basic]
In Windows 98 and Windows 2000, the speaker configuration can be set by the user in Control Panel. An application can retrieve this value by using **DirectSound8.GetSpeakerConfig** and override it by calling **DirectSound8.SetSpeakerConfig**.

There is no guarantee that **SetSpeakerConfig** will have an immediate effect; it might simply change a registry setting that will not take effect until the computer is restarted. Instead of using this method, applications should advise the user to change the speaker configuration through Control Panel.

In Windows 95, **GetSpeakerConfig** simply returns a default value or the last value you set with **SetSpeakerConfig**.

Compacting Hardware Memory

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
As long as it has at least the priority cooperative level, your application can use the **IDirectSound8::Compact** method to move any on-board sound memory into a contiguous block to make the largest portion of free memory available.

DirectSound Buffers

DirectSound buffer objects control the delivery of wave data from a source to a destination. The source might be a synthesizer, another buffer, a file, or a resource. For most buffers, the destination is a mixing engine called the primary buffer. From the primary buffer, the data goes to the hardware that converts the PCM samples to sound waves.

The information in this section is of interest mainly for applications originally written for previous versions of DirectSound, or applications that need to have low-level control over buffers. If your application is playing sounds by using the DirectMusic performance object, buffer creation and management is done automatically. You can, however, obtain buffer objects in order to gain access to parameters for 3-D effects and effects in DMOs. For more information, see Using Effects and 3-D Sound.

Information about using DirectSound buffers is contained in the following topics:

- Buffer Basics
- Creating Secondary Buffers
- Duplicating Buffers
- Buffer Control Options
- 3-D Algorithms for Buffers
- Filling and Playing Static Buffers
- Using Streaming Buffers
- Playback Controls
- Play and Write Cursors
- Play Buffer Notification
- Mixing Sounds
- Buffer Management

For information about capture buffers, see Capturing Waves.

Buffer Basics

When DirectSound is initialized, it automatically creates and manages a primary sound buffer for mixing sounds and sending them to the output device.

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

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

Buffers can be located either in hardware or in software. Hardware buffers are mixed by the sound card processor, and software buffers are mixed by the CPU. Software buffer data is always in system memory; hardware buffer data can be in system memory or, if the application requests it and resources are available, in on-board memory. For more information, see Dynamic Voice Management and Hardware Acceleration on ISA and PCI Cards.

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

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

Creating Secondary Buffers

[C++]
To create a sound buffer, call the **IDirectSound8::CreateSoundBuffer** method. This method returns a pointer to an **IDirectSoundBuffer8** interface, which the application uses to manipulate and play the buffer.

The following example shows how to create a secondary sound buffer:

```
BOOL AppCreateBasicBuffer(
      LPDIRECTSOUND8 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));
   dsbdesc.dwSize = sizeof(DSBUFFERDESC);
   dsbdesc.dwFlags =
      DSBCAPS_CTRLPAN | DSBCAPS_CTRLVOLUME | DSBCAPS_CTRLFREQUENCY;
```

```
    dsbdesc.dwBufferBytes = 3 * pcmwf.wf.nAvgBytesPerSec;
    dsbdesc.lpwfxFormat = (LPWAVEFORMATEX)&pcmwf;

    // Create buffer.

    hr = lpDirectSound->CreateSoundBuffer(&dsbdesc, lplpDsb, NULL);
    if SUCCEEDED(hr)
    {
        // IDirectSoundBuffer interface is in *lplpDsb.
        // Use QueryInterface to obtain IDirectSoundBuffer8.
        return TRUE;
    }
    else
    {
        // Failed.
        *lplpDsb = NULL;
        return FALSE;
    }
}
```

The example creates a streaming buffer large enough to hold 3 seconds of streaming data. Nonstreaming buffers should be made just large enough to accommodate the entire sound.

DirectSound allocates hardware resources to the first buffer that can take advantage of them. Because hardware buffers are mixed by the sound card processor, they have much less impact on application performance.

If you wish to specify the location of a buffer rather than letting DirectSound decide where it belongs, set either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag in the **DSBUFFERDESC** structure. If the DSBCAPS_LOCHARDWARE flag is set and there are insufficient hardware resources, the buffer creation request fails.

To take advantage of the voice management features of DirectSound, specify the DSBCAPS_LOCDEFER flag when creating the buffer. This flag defers the allocation of resources for the buffer until it is played. For more information, see Dynamic Voice Management.

You can ascertain the location of an existing buffer by using the **IDirectSoundBuffer8::GetCaps** method and checking the **dwFlags** member of the **DSBCAPS** structure for either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags. One or the other is always specified.

Setting the DSBCAPS_STATIC flag lets DirectSound know that the buffer should be created in on-board hardware memory if possible. The **IDirectSound::CreateSoundBuffer** method does not fail if a hardware buffer is not available. This flag has no effect on most modern sound cards, which use system memory for their buffers. Hardware static buffers should be used only for short

sounds that are to be played repeatedly. For more information, see Voice Management on ISA and PCI Cards.

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

[Visual Basic]
To create a secondary sound buffer, call the **DirectSound8.CreateSoundBuffer**, **DirectSound8.CreateSoundBufferFromFile**, or **DirectSound8.CreateSoundBufferFromResource** method. These methods create a **DirectSoundSecondaryBuffer8** object, which can then be used to manipulate and play the buffer.

CreateSoundBufferFromFile and **CreateSoundBufferFromResource** are good methods to use when the sound is not very long. DirectSound creates a buffer just big enough to hold the entire sound, matches its wave format to the format of the data, and fills it with the data. **CreateSoundBuffer** is suitable for streaming buffers; however, you have to do your own parsing of the wave file or resource as well as handling the streaming of data to the buffer.

Always specify the DSBCAPS_GETCURRENTPOSITION2 flag in the **DSBUFFERDESC** type when creating a streaming buffer.

The following example shows how to create a streaming buffer able to hold 2 seconds of data. Assume that the *m_ds* variable has been set to a **DirectSound8** object.

```
Dim m_dsb As DirectSoundSecondaryBuffer8

Public Sub CreateSoundBuffer()

  Dim buffdesc As DSBUFFERDESC

  ' Describe the sound buffer to be created.

  With buffdesc
    lFlags = DSBCAPS_CTRLPAN Or DSBCAPS_CTRLVOLUME _
        Or DSBCAPS_CTRLFREQUENCY

    With .fxFormat
      .nFormatTag = WAVE_FORMAT_PCM
      .nChannels = 2
      .lSamplesPerSec = 22050
      .nBitsPerSample = 16
      .nBlockAlign = (.nChannels * .nBitsPerSample) / 8
      .lAvgBytesPerSec = .lSamplesPerSec * .nBlockAlign
      .nSize = 0    ' Ignored for WAVE_FORMAT_PCM
    End With
```

```
    lBufferBytes = .fxFormat.lAvgBytesPerSec * 2

  End With

  Set m_dsb = ds.CreateSoundBuffer(buffdesc)

End Sub
```

Creating a buffer from a file or resource is much simpler, as the following example shows:

```
Dim m_dsb As DirectSoundSecondaryBuffer8
Dim buffdesc As DSBUFFERDESC

buffdesc.lFlags = DSBCAPS_CTRLPAN Or DSBCAPS_CTRLVOLUME _
  Or DSBCAPS_CTRLFREQUENCY
Set dsbuffer = ds.CreateSoundBufferFromFile( _
  "c:\media\ding.wav", dsbdesc)
```

In this case, the format of the buffer is based on that of the data, and the method places this information in *dsbdesc.fxFormat*.

DirectSound allocates hardware resources to the first buffer that can take advantage of them. Because hardware buffers are mixed by the sound card processor, they have much less impact on application performance.

If you wish to specify the location of a buffer rather than letting DirectSound decide where it belongs, set either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag in the **DSBUFFERDESC** type. If the DSBCAPS_LOCHARDWARE flag is set and there are insufficient hardware resources, the buffer creation request fails.

To take advantage of the voice management features of DirectSound, specify the DSBCAPS_LOCDEFER flag when creating the buffer. This flag defers the allocation of resources for the buffer until it is played. For more information, see Dynamic Voice Management.

You can ascertain the location of an existing buffer by using the **DirectSoundSecondaryBuffer8.GetCaps** method and checking the **lFlags** member of the **DSBCAPS** type for either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags. One or the other is always specified.

Setting the DSBCAPS_STATIC flag lets DirectSound know that the buffer should be created in on-board hardware memory if possible. The buffer creation method does not fail if a hardware buffer is not available. This flag has no effect on most modern sound cards, which use system memory for their buffers. Hardware static buffers should be used only for short sounds that are to be played repeatedly. For more information, see Voice Management on ISA and PCI Cards.

Duplicating Buffers

[C++]
You can create two or more secondary buffers containing the same data by using the **IDirectSound8::DuplicateSoundBuffer** method. You cannot duplicate the primary sound buffer. There is no guarantee that the duplicate buffer actually shares memory with the original buffer, so it is unwise to make changes to one buffer in the expectation that these changes will be reflected in all duplicate buffers.

[Visual Basic]
You can create two or more secondary buffers containing the same data by using the **DirectSound8.DuplicateSoundBuffer** method. You cannot duplicate the primary sound buffer. There is no guarantee that the duplicate buffer actually shares memory with the original buffer, so it is unwise to make changes to one buffer in the expectation that these changes will be reflected in all duplicate buffers.

Buffer Control Options

[C++]
When creating a sound buffer, your application must specify the control options needed for that buffer. This is done with the **dwFlags** member of the **DSBUFFERDESC** structure, which can contain one or more DSBCAPS_CTRL* flags.

The following controls are available:

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

[Visual Basic]
When creating a sound buffer, your application must specify the control options needed for that buffer. This is done with the **lFlags** member of the **DSBUFFERDESC** type, which can contain one or more DSBCAPS_CTRL* flags.

The following controls are available:

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

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

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

Note that certain combinations of controls are not allowed. For example, pan and 3-D are mutually exclusive.

[C++]
If your application attempts to use a control that a buffer lacks, the method call fails. For example, if you attempt to change the volume by using the **IDirectSoundBuffer8::SetVolume** method, the method can succeed only if the DSBCAPS_CTRLVOLUME flag was specified when the buffer was created. Otherwise the method fails and returns the DSERR_CONTROLUNAVAIL error code.

[Visual Basic]
If your application attempts to use a control that a buffer lacks, the method call fails. For example, if you attempt to change the volume by using the **DirectSoundSecondaryBuffer8.SetVolume** method, the method can succeed only if the DSBCAPS_CTRLVOLUME flag was specified when the buffer was created. Otherwise, the method fails and raises the DSERR_CONTROLUNAVAIL error.

See also Playback Controls.

3-D Algorithms for Buffers

When you create a secondary buffer with the DSBCAPS_CTRL3D control flag, you can also specify an algorithm to be used in spatializing the sound if the buffer is in software. By default, no HRTF processing is performed, and the location of the sound relative to the listener is indicated by panning and volume only. You can request two levels of HRTF for the buffer.

[C++]
For more information, see **DSBUFFERDESC**.

[Visual Basic]
For more information, see **DSBUFFERDESC**.

Filling and Playing Static Buffers

A secondary buffer that contains an entire self-contained sound is called a static buffer. Although it is possible to reuse the same buffer for different sounds, typically data is written to a static buffer only once.

Static buffers are created and managed just like streaming buffers. The only difference is in the way they are used: static buffers are filled once and then played, but streaming buffers are constantly refreshed with data as they are playing.

Note

A static buffer is not necessarily one created by setting the DSBCAPS_STATIC flag in the buffer description. This flag requests allocation of memory on the sound card, which not available on most modern hardware. A static buffer can exist in system memory and can be created with either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag. For more information, see Voice Management on ISA and PCI Cards.

[C++]
Loading data into a static buffer is a three-step process:

1. Lock the entire buffer by using **IDirectSoundBuffer8::Lock**. You specify the offset within the buffer where you intend to begin writing (normally 0), and get back the memory address of that point.
2. Write the audio data to the returned address by using a standard memory-copy routine.
3. Unlock the buffer using **IDirectSoundBuffer8::Unlock**.

These steps are shown in the following example, where *lpdsbStatic* is an **IDirectSoundBuffer8** interface pointer and *pbData* is the address of the data source:

```
LPVOID lpvWrite;
DWORD  dwLength;

if (DS_OK == lpdsbStatic->Lock(
      0,          // Offset at which to start lock.
      0,          // Size of lock; ignored because of flag.
      &lpvWrite,     // Gets address of first part of lock.
```

```
        &dwLength,      // Gets size of first part of lock.
        NULL,           // Address of wraparound not needed.
        NULL,           // Size of wraparound not needed.
        DSBLOCK_ENTIREBUFFER))  // Flag.
{
    memcpy(lpvWrite, pbData, dwLength);
    lpdsbStatic->Unlock(
        lpvWrite,       // Address of lock start.
        dwLength,       // Size of lock.
        NULL,           // No wraparound portion.
        0);             // No wraparound size.
}
```

To play the buffer, call **IDirectSoundBuffer8::Play**, as in the following example:

```
lpdsbStatic->SetCurrentPosition(0);
HRESULT hr = lpdsbStatic->Play(
        0,    // Unused.
        0,    // Priority for voice management.
        0);   // Flags.
```

Because the DSBPLAY_LOOPING flag is not set in the example, the buffer automatically stops when it reaches the end. You can also stop it prematurely by using **IDirectSoundBuffer8::Stop**. When you stop a buffer prematurely, the play cursor position remains where it is. Hence the call to **IDirectSoundBuffer8::SetCurrentPosition** in the example, which ensures that the buffer starts from the beginning.

[Visual Basic]

A static buffer can be created and filled with data by using a single method call, either **DirectSound8.CreateSoundBufferFromFile** or **DirectSound8.CreateSoundBufferFromResource**.

The following sample code creates a static buffer from the wave identified as "bounce" in the resource file. Assume that *m_ds* is a **DirectSound8** object.

```
Dim dsbd As DSBUFFERDESC
Dim dsbBounce As DirectSoundSecondaryBuffer8

dsbd.lFlags = DSBCAPS_CTRLVOLUME
Set dsbBounce = m_ds.CreateSoundBufferFromResource( _
        vbNullString, "bounce", dsbd)
```

The resource file has been compiled into the executable file, so no module name needs to be provided. Nor does the value of the **lBufferBytes** member of the **DSBUFFERDESC** type need to be set, because the method determines the size of the

buffer from the size of the data. The **WAVEFORMATEX** type receives information about the wave format from the header stored in the resource.

To play the buffer, call **DirectSoundSecondaryBuffer8.Play**, as in the following example:

```
dsbResource.SetCurrentPosition 0
dsbResource.Play 0
```

Because the DSBPLAY_LOOPING flag is not set in the example, the buffer automatically stops when it reaches the end. You can also stop it prematurely by using **DirectSoundSecondaryBuffer8.Stop**. When you stop a buffer prematurely, the play cursor remains where it is. Hence the call to **DirectSoundSecondaryBuffer8.SetCurrentPosition** in the example, which ensures that the buffer starts from the beginning.

You can also create an empty static buffer by using **DirectSound8.CreateSoundBuffer** and then fill it with data from another location in memory by using the **DirectSoundSecondaryBuffer8.WriteBuffer** method.

More information on writing to secondary buffers is given in the next topic, Using Streaming Buffers.

Using Streaming Buffers

A streaming buffer plays a long sound that cannot all fit into the buffer at once. As the buffer plays, old data is periodically replaced with new data.

[C++]
To play a streaming buffer, call the **IDirectSoundBuffer8::Play** method, specifying DSBPLAY_LOOPING in the *dwFlags* parameter.

To halt playback, call the **IDirectSoundBuffer8::Stop** method. This method stops the buffer immediately, so you need to be sure that all data has been played. This can be done by polling the play position or by setting a notification position.

Streaming into a buffer requires the following steps:

1. Lock a portion of the buffer by using **IDirectSoundBuffer8::Lock**. This method returns one or two addresses where data can now be written.
2. Write the audio data to the returned address or addresses by using a standard memory-copy routine.
3. Unlock the buffer using **IDirectSoundBuffer8::Unlock**.

The reason **IDirectSoundBuffer8::Lock** might return two addresses is that you can lock any number of bytes, up to the size of the buffer, regardless of the start point. If necessary, the locked portion wraps around to the beginning of the buffer. If it does, you have to perform two separate memory copies.

For example, say you lock 30,000 bytes beginning at offset 20,000 in a 40,000-byte buffer. When you call **Lock** in this case, it returns four values:

- The address of offset 20,000.
- The number of bytes locked from that point to the end of the buffer (20,000). You write this number of bytes to the first address.
- The address of offset 0.
- The number of bytes locked from that point (10,000). You write this number of bytes to the second address.

If no wraparound is necessary, the last two values are NULL and 0 respectively.

Although it's possible to lock the entire buffer, you must not do so while it is playing. Refresh only a portion of the buffer each time. For example, you might lock and write to the first quarter of the buffer as soon as the play cursor reaches the second quarter, and so on. You must never write to the part of the buffer that lies between the play cursor and the write cursor. For more information, see Play and Write Cursors.

The following function writes data to a sound buffer, starting at the position passed in *dwOffset*:

```
BOOL AppWriteDataToBuffer(
    LPDIRECTSOUNDBUFFER8 lpDsb,  // The buffer.
    DWORD dwOffset,              // Our own write cursor.
    LPBYTE lpbSoundData,         // Start of our data.
    DWORD dwSoundBytes)          // Size of block to copy.
{
    LPVOID  lpvPtr1;
    DWORD   dwBytes1;
    LPVOID  lpvPtr2;
    DWORD   dwBytes2;
    HRESULT hr;

    // Obtain memory address of write block. This will be in two parts
    // if the block wraps around.

    hr = lpDsb->Lock(dwOffset, dwSoundBytes, &lpvPtr1,
        &dwBytes1, &lpvPtr2, &dwBytes2, 0);

    // If the buffer was lost, restore and retry lock.

    if (DSERR_BUFFERLOST == hr)

    {
        lpDsb->Restore();
        hr = lpDsb->Lock(dwOffset, dwSoundBytes,
            &lpvPtr1, &dwBytes1, &lpvPtr2, &dwBytes2, 0);
```

```
    }
    if SUCCEEDED(hr)
    {
        // Write to pointers.

        CopyMemory(lpvPtr1, lpbSoundData, dwBytes1);
        if (NULL != lpvPtr2)
        {
            CopyMemory(lpvPtr2, lpbSoundData+dwBytes1, dwBytes2);
        }

        // Release the data back to DirectSound.

        hr = lpDsb->Unlock(lpvPtr1, dwBytes1, lpvPtr2,
            dwBytes2);
        if SUCCEEDED(hr)
        {
            // Success.
            return TRUE;
        }
    }

    // Lock, Unlock, or Restore failed.

    return FALSE;
}
```

[Visual Basic]

To play a streaming buffer, call the **DirectSoundSecondaryBuffer8.Play** method, specifying DSBPLAY_LOOPING in the *flags* parameter.

To halt playback, call the **DirectSoundSecondaryBuffer8.Stop** method. This method stops the buffer immediately, so you need to be sure that all data has been played. This can be done by polling the play position or by setting a notification position.

Stream data into a buffer by using the **DirectSoundSecondaryBuffer8.WriteBuffer** method. This method lets you write data to any portion of the buffer. Although it's possible to write to the entire buffer, you must not do so while it is playing. Refresh only a portion of the buffer each time. For example, you might lock and write to the first quarter of the buffer as soon as the play cursor reaches the second quarter, and so on. You must never write to the part of the buffer that lies between the play cursor and the write cursor. For more information, see Play and Write Cursors.

The following sample codewrites 1000 bytes of data from the byte array *myBuffer* to the beginning of the buffer represented by *dsb*:

```
dsb.WriteBuffer 0, 1000, myBuffer(0), DSBLOCK_DEFAULT
```

WriteBuffer handles wraparound transparently. Suppose the secondary buffer has been created with a size of 10,000 bytes, and the application writes 5000 bytes to an offset past the midpoint of the buffer, as follows:

```
dsb.WriteBuffer 8000, 5000, myBuffer(0), DSBLOCK_DEFAULT
```

This call writes the first 2000 bytes to the last part of the buffer and the next 3000 bytes to the beginning of the buffer.

It is the application's responsibility to track the offset for the next data write. The easiest way to do this is by setting notification positions and writing a fixed number of bytes each time an event is triggered.

For more information, see the following topics:

- Play and Write Cursors
- Play Buffer Notification
- Reading Wave Files

Playback Controls

[C++]
To retrieve and set the volume at which a buffer is played, your application can use the **IDirectSoundBuffer8::GetVolume** and **IDirectSoundBuffer8::SetVolume** methods. Setting the volume on the primary sound buffer changes the waveform-audio volume of the sound card.

By calling the **IDirectSoundBuffer8::GetFrequency** and **IDirectSoundBuffer8::SetFrequency** methods, you can retrieve and set the frequency at which audio samples play. You cannot change the frequency of the primary buffer.

To retrieve and set the pan, you can call the **IDirectSoundBuffer8::GetPan** and **IDirectSoundBuffer8::SetPan** methods.

[Visual Basic]
To retrieve and set the volume at which a buffer is played, your application can use the **DirectSoundSecondaryBuffer8.GetVolume** and **DirectSoundSecondaryBuffer8.SetVolume** methods. Equivalent methods are available on the **DirectSoundPrimaryBuffer8** object. Setting the volume on the primary sound buffer changes the waveform-audio volume of the sound card.

By calling the **DirectSoundSecondaryBuffer8.GetFrequency** and **DirectSoundSecondaryBuffer8.SetFrequency** methods, you can retrieve and set the frequency at which audio samples play. You cannot change the frequency of the primary buffer.

To retrieve and set the pan, use the **DirectSoundSecondaryBuffer8.GetPan** and **DirectSoundSecondaryBuffer8.SetPan** methods. Equivalent methods are available on the **DirectSoundPrimaryBuffer8** object.

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

Play and Write Cursors

DirectSound maintains two pointers into the buffer: the play cursor and the write cursor. These positions are byte offsets into the buffer, not absolute memory addresses.

[C++]
The **IDirectSoundBuffer8::Play** method always starts playing at the buffer's play cursor. When a buffer is created, the cursor position is set to zero. As a sound is played, the cursor moves and always points to the next byte of data to be output. When the buffer is stopped, the cursor remains at the next byte of data.

[Visual Basic]
The **DirectSoundSecondaryBuffer8.Play** method always starts playing at the buffer's play cursor. When a buffer is created, the cursor position is set to zero. As a sound is played, the cursor moves and always points to the next byte of data to be output. When the buffer is stopped, the cursor remains at the next byte of data.

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

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

The write position moves with the play position, not with data written to the buffer. If you're streaming data, you are responsible for maintaining your own pointer into the buffer to indicate where the next block of data should be written.

[C++]
An application can retrieve the play and write cursors by calling the **IDirectSoundBuffer8::GetCurrentPosition** method. The **IDirectSoundBuffer8::SetCurrentPosition** method lets you move the play cursor. Applications do not control the position of the write cursor.

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

[Visual Basic]
An application can retrieve the play and write cursors by calling the **DirectSoundSecondaryBuffer8.GetCurrentPosition** method. The **DirectSoundSecondaryBuffer8.SetCurrentPosition** method lets you move the play cursor. Applications do not control the position of the write cursor.

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

Play Buffer Notification

[C++]
When streaming audio, you may want your application to be notified when the play cursor reaches a certain point in the buffer, or when playback is stopped. With the **IDirectSoundNotify8::SetNotificationPositions** method, you can set any number of points within the buffer where events are to be signaled. You cannot do this while the buffer is playing.

First you have to obtain a pointer to the **IDirectSoundNotify8** interface. You can do this by using the buffer object's **QueryInterface** method, as in the following example, where *lpDsbSecondary* is an **IDirectSoundBuffer8** interface pointer:

```
LPDIRECTSOUNDNOTIFY8 lpDsNotify;

HRESULT hr = lpDsbSecondary->QueryInterface(IID_IDirectSoundNotify8,
                                (LPVOID *)&lpDsNotify);
if SUCCEEDED(hr)
{
   // Go ahead and use lpDsNotify->SetNotificationPositions.
}
```

The **IDirectSoundNotify8** interface is associated with the buffer object from which you obtained the pointer. The methods of the interface automatically apply to that buffer.

Now create an event object with the Win32 **CreateEvent** function. Put the handle to this event in the **hEventNotify** member of a **DSBPOSITIONNOTIFY** structure. In the **dwOffset** member of that structure, specify the offset within the buffer where you want the event to be signaled. Then pass the address of the structure—or of an array

of structures, if you want to set more than one notification position—to the **IDirectSoundNotify8::SetNotificationPositions** method.

The following example sets a single notification position. The event will be signaled when playback stops, either because it was not looping and the end of the buffer has been reached, or because the application called the **IDirectSoundBuffer8::Stop** method.

```
DSBPOSITIONNOTIFY PositionNotify;

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

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

If you are taking advantage of voice management, it's possible that a buffer could be terminated before a notification position is reached. See the Remarks for **DSBPOSITIONNOTIFY**.

[Visual Basic]
When streaming audio, you may want your application to be notified when the play cursor reaches a certain point in the buffer, or when playback is stopped. By using the **DirectSoundSecondaryBuffer8.SetNotificationPositions** method, you can set any number of points within the buffer where events are to be signaled. You cannot do this while the buffer is playing.

To set notification events on a buffer, you must first implement the **DirectXEvent8** class in the form that contains the code for the buffer. Create a **DirectXEvent8.DXCallback** method by choosing **DirectXEvent8** from the drop-down class list, and add the code you want to be executed when the event is signaled.

Create an event handle by calling the **DirectX8.CreateEvent** method and pass the event handle in the **hEventNotify** member of a **DSBPOSITIONNOTIFY** type. Pass the type to the **DirectSoundSecondaryBuffer8.SetNotificationPositions** method. The event handle will now be passed to **DirectXEvent8.DXCallback** whenever the play cursor reaches that position.

If you are taking advantage of voice management, it's possible that a buffer could be terminated before a notification position is reached. See the Remarks for **DSBPOSITIONNOTIFY**.

The following example code shows how to set up a single notification position. The following are the relevant lines in the Declarations section of the module:

```
Implements DirectXEvent8

Dim gDX As New DirectX8
Dim gDSB As DirectSoundSecondaryBuffer8
```

```
Dim endEvent As Long
Dim dsbpn(0) As DSBPOSITIONNOTIFY
```

Note that *dsbpn* must be an array, even though it contains only a single notification position.

In the **Form_Load** subroutine, an event is created:

```
endEvent = gDX.CreateEvent(Me)
```

After the **DirectSoundSecondaryBuffer8** object has been created, the notification position is set and associated with the event, as follows:

```
With dsbpbn(0)
  .hEventNotify = endEvent
  .lOffset = DSBPN_OFFSETSTOP
End With
gDSB.SetNotificationPositions 1, dsbpn()
```

The offset of the notification, DSBPN_OFFSETSTOP or -1, is a special value indicating that the event is to be set when the buffer stops playing, either because it was not looping and the end of the buffer has been reached, or because the application called the **DirectSoundSecondaryBuffer8.Stop** method. When this happens, the application-defined callback function is automatically called, with the event identifier *endEvent* being passed in as the parameter.

```
Private Sub DirectXEvent8_DXCallback(ByVal eventid As Long)
  If eventid = endEvent Then
    lblStatus.Caption = "Buffer stopped."
  End If
End Sub
```

Mixing Sounds

[C++]
It is easy to mix multiple streams with DirectSound. You simply create secondary sound buffers, obtaining an **IDirectSoundBuffer8** interface for each sound. You then play the buffers simultaneously. DirectSound takes care of the mixing in the primary sound buffer and plays the result.

[Visual Basic]
It is easy to mix multiple streams with DirectSound. You simply create secondary sound buffers, obtaining a **DirectSoundSecondaryBuffer8** object for each sound. You then play the buffers simultaneously. DirectSound takes care of the mixing in the primary sound buffer and plays the result.

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

[C++]
Your application can change the hardware output format by creating a primary sound buffer object and calling the **IDirectSoundBuffer8::SetFormat** method. This primary buffer object is for control purposes only; creating it is not the same as obtaining write access to the primary buffer as described under Writing to the Primary Buffer, and you do not need the DSSCL_WRITEPRIMARY cooperative level. However, you do need a cooperative level of DSSCL_PRIORITY or higher in order to call the **SetFormat** method. DirectSound will restore the hardware format to the format specified in the last call every time the application gains the input focus.

[Visual Basic]
Your application can change the hardware output format by creating a primary sound buffer and calling the **DirectSoundPrimaryBuffer8.SetFormat** method. You need a cooperative level of DSSCL_PRIORITY or higher in order to call the **SetFormat** method. DirectSound will restore the hardware format to the format specified in the last call every time the application gains the input focus.

You must set the format of the primary buffer before creating any secondary buffers.

Note
With WDM drivers, setting the primary buffer format has no effect. The format is determined by the kernel mixer. For more information, see DirectSound Driver Models.

Buffer Management

[C++]
The **IDirectSoundBuffer8::GetCaps** method retrieves the capabilities of the DirectSoundBuffer object.

Your application can use the **IDirectSoundBuffer8::GetStatus** method to determine if the buffer is playing or if it has stopped.

Use the **IDirectSoundBuffer8::GetFormat** method to retrieve information about the format of the sound data in a buffer. You also can use **IDirectSoundBuffer8::SetFormat** to set the format of the sound data in the primary buffer. For more information, see Mixing Sounds.

Note

The **SetFormat** method cannot be called on secondary buffers. After a secondary buffer is created, its format is fixed. If you need a secondary buffer that uses another format, you must create a new sound buffer with this format.

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

The DSERR_BUFFERLOST error code is returned when the **IDirectSoundBuffer8::Lock** or **IDirectSoundBuffer8::Play** method is called for a lost buffer. When the application that caused the loss either lowers its cooperative level from write-primary or moves to the background, other applications can attempt to reallocate the buffer memory by calling the **IDirectSoundBuffer8::Restore** method. If successful, this method restores the buffer memory and all other settings for the buffer, such as volume and pan settings. However, a restored buffer may not contain valid sound data, so the owning application should rewrite the data to the buffer.

[Visual Basic]
The **DirectSoundPrimaryBuffer8.GetCaps** and **DirectSoundSecondaryBuffer8.GetCaps** methods retrieve the capabilities of a buffer.

Your application can use the **DirectSoundPrimaryBuffer8.GetStatus** or **DirectSoundSecondaryBuffer8.GetStatus** method to determine if the buffer is playing or if it has stopped.

You can use the **DirectSoundPrimaryBuffer8.GetFormat** or **DirectSoundSecondaryBuffer8.GetFormat** method to retrieve information about the format of the sound data in the buffer. You also can use the **DirectSoundPrimaryBuffer8.SetFormat** method to retrieve and set the format of the sound data in the primary buffer.

Note

There is no **SetFormat** method for **DirectSoundSecondaryBuffer8**. After a secondary buffer is created, its format is fixed. If you need a secondary buffer that uses another format, you must create a new sound buffer with this format.

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

The DSERR_BUFFERLOST error is raised when the **DirectSoundSecondaryBuffer8.WriteBuffer** or **DirectSoundSecondaryBuffer8.Play** method is called for a lost buffer. When the application that caused the loss either lowers its cooperative level from write-primary or moves to the background, other applications can attempt to reallocate the buffer memory by calling the **DirectSoundSecondaryBuffer8.Restore** method. If successful, this method restores the buffer memory and all other settings for the buffer, such as volume and pan settings. However, a restored buffer may not contain valid sound data, so the owning application should rewrite the data to the buffer.

Using Wave Data

DirectSound buffers play only waveform audio data, which consists of digital samples of the sound at a fixed sampling rate. The representation of an analog signal by a sequence of numbers is known as pulse code modulation (PCM).

Wave data is usually stored in files or resources in Resource Interchange File Format (RIFF). The data includes a description of the wave format, including parameters such as the sampling rate and number of output channels.

[C++]
The particular format of a sound can be described by a **WAVEFORMATEX** or **WAVEFORMATEXTENSIBLE** structure.

The **wFormatTag** member of this structure contains a unique identifier assigned by Microsoft Corporation. The only tags valid with DirectSound are the following:

- WAVE_FORMAT_PCM. This tag indicates pulse code modulation (PCM), an uncompressed format.
- WAVE_FORMAT_EXTENSIBLE. Multichannel audio format; valid only with WDM drivers. For more information, see Multichannel Wave Formats.

[Visual Basic]
The particular format of a sound can be described by a **WAVEFORMATEX** type.

Note

The **WAVEFORMATEX** type in the DirectX for Visual Basic type library is not identical to the **WAVEFORMATEX** structure declared in Mmreg.h and used in the format chunk of a wave file or resource. For information on how to read a format chunk and copy the relevant data into a DirectX **WAVEFORMATEX** type, see Reading Wave Files.

When you are creating static buffers by using **DirectSound8.CreateSoundBufferFromFile** or **DirectSound8.CreateSoundBufferFromResource**, you do not specify a buffer

format, because the format of the created buffer always matches the format of the data. You do, however, have to specify the format of any buffer created by using **DirectSound8.CreateSoundBuffer**, and this format must match the format of the data you intend to play in the buffer.

DirectSound does not support compressed wave formats. Applications should use the audio compression manager (ACM) functions, provided with the Win32 APIs in the Platform SDK, to convert compressed audio to PCM format before writing the data to a sound buffer. Alternatively, use the DirectMusic loader, which automatically enlists the help of ACM when loading compressed waves for playback through the DirectMusic performance.

More information on waves is included in the following topics:

- Multichannel Wave Formats
- Reading Wave Files
- Reading Wave Data from a Resource

Multichannel Wave Formats

[C++]
On WDM drivers, DirectSound buffers support wave formats that have more than two output channels, for speaker configurations such as 5.1, which has speakers at the front left, front center, front right, back left, and back right, plus a low-frequency enhancer.

The **WAVEFORMATEXTENSIBLE** structure describes a multichannel wave format. This structure is an extension of **WAVEFORMATEX** that configures the extra bytes already supported by the **cbSize** member of **WAVEFORMATEX**. A **WAVEFORMATEXTENSIBLE** structure can be cast as **WAVEFORMATEX** wherever the latter is expected, as for example in the **DSBUFFERDESC** structure.

If there are fewer physical speakers than the number of channels specified in a multichannel wave file, the audio data is mixed appropriately and output to the existing speakers.

DirectSound does not support effects or 3-D processing on buffers in a multichannel format. An attempt to create a buffer with the DSBCAPS_CTRL3D or DSBCAPS_CTRLFX flag and a multichannel wave format will fail.

For more information on multichannel wave formats, see "Enhanced Audio Formats for Multi-Channel Configurations and High-Bit Resolution", available at http://www.microsoft.com/hwdev/audio/multichaud.htm.

[Visual Basic]

This topic pertains only to applications written in C++. DirectX for Visual Basic does not support wave formats with more than two channels.

Reading Wave Files

This topic explains how to load data from wave files into DirectSound buffers without the help of the DirectMusic loader and performance. Most applications do not need to work directly with wave data and buffers. For the preferred method of loading and playing waves, see Loading Audio Data and Playing Sounds.

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

The DirectSound API does not include methods for handling wave files. However, the Dsutil.cpp file used by many of the SDK sample applications implements a **CWaveFile** class that has the following public methods:

- **Open**. Opens a file for reading and retrieves the wave format, or opens the file for writing and writes the header chunks.
- **GetSize**. Returns the size of the data chunk, after **Open** has been called.
- **Read**. Copies a portion of the data chunk into a buffer and advances the read cursor.
- **Write**. Writes from a buffer to the data chunk and advances the write cursor.
- **ResetFile**. Sets the read and write cursors to the beginning of the data chunk.
- **Close**. Closes the file.

The first step in reading a wave file is to call the **CWaveFile::Open** method. This verifies that the file is in RIFF format and gets information about the wave format. The parameters are the filename, NULL for the format, and the WAVEFILE_READ flag. The method returns an **HRESULT**.

The following code opens a wave file:

```
CWaveFile waveFile;

if (FAILED(waveFile.Open("mywave.wav", NULL, WAVEFILE_READ)))
{
    waveFile.Close();
}
```

The application can now begin copying data from the file to a secondary sound buffer. Normally you don't create the sound buffer until you have obtained the format

of the wave. The following code creates a static buffer just large enough to hold all the data in the file:

```
LPDIRECTSOUNDBUFFER  lpdsbStatic;
DSBUFFERDESC         dsbdesc;

memset(&dsbdesc, 0, sizeof(DSBUFFERDESC));
dsbdesc.dwSize = sizeof(DSBUFFERDESC);
dsbdesc.dwFlags = 0;

/* The wave format and size of the data chunk are stored in
// CWaveFile after CWaveFile::Open has been called.

dsbdesc.dwBufferBytes = waveFile.GetSize();
dsbdesc.lpwfxFormat = waveFile.m_pwfx;

// lpds is a valid IDirectSound8 pointer.

if FAILED(lpds->CreateSoundBuffer(&dsbdesc, &lpdsbStatic, NULL))
{
   // Error handling.
   .
   .
   .
}
```

Now the buffer can be filled with data. Because in this case the application is not streaming the data, the entire buffer is locked from the beginning. There is no wraparound, so only a single pointer and byte count are required.

```
LPVOID lpvAudio1;
DWORD  dwBytes1;

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

The **CWaveFile::Read** method can then be used to fill the buffer, as follows:

```
DWORD dwBytesRead;

waveFile.Read((BYTE*)lpvAudio1, dwBytes1, &dwBytesRead);
```

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

```
lpdsbStatic->Unlock(lpvAudio1, dwBytes1, NULL, 0);
waveFile.Close();
```

For a streaming buffer, you would typically call **CWaveFile::Read** at regular intervals determined by the movement of the play cursor. If the locked portion of the buffer wraps around, call the method once for each segment of the lock.

For more information on streaming, see Using Streaming Buffers and Play Buffer Notification.

See the sample applications in the DirectX SDK installation for more examples of using the **CWaveFile** class.

[Visual Basic]
For short sounds, the easiest way to play a wave file using DirectSound alone is to load it into a static buffer by calling the **DirectSound8.CreateSoundBufferFromFile** method. See Creating Secondary Buffers.

For larger files that will not conveniently fit in memory, you need to create a streaming buffer and read the file in pieces.

Wave files are in the Resource Interchange File Format (RIFF), which consists of a file header followed by variable number of "chunks," each made up of a header and data. The chunk header consists of a four-character tag identifying the type of data and a **Long** giving the length of the data.

The wave file header is organized as follows:

- The string "RIFF", which can also be read as the **Long** &H46464952.
- A **Long** giving the size of the file.
- The string "WAVE", which can also be read as the **Long** &H45564157.

The first chunk is always the format chunk, which looks like this:

- The string "fmt" followed by a space.
- A **Long** giving the size of the format data.
- Data describing the wave format of the data. For files in PCM format, this is 16 bytes of data equivalent to that in the **WAVEFORMATEX** type.

The PCM sample data in a wave file is contained in a chunk beginning with the string "data", which can also be read as the **Long** &H61746164. Most often this chunk immediately follows the format chunk, but because RIFF is an extensible format, there is no guarantee that some other type of chunk will not precede it. Your file parser must be capable of ignoring chunks it cannot handle.

To parse a wave file, it is helpful to have three user-defined types. The first will receive all the information in the file header and in the header of the format chunk.

```
Private Type FileHeader
   lRiff As Long
   lFileSize As Long
   lWave As Long
   lFormat As Long
   lFormatLength As Long
End Type
```

The second type will receive the format data. You can't use the **WAVEFORMATEX** type for this, because the members are in a different order. Your user-defined type needs to retrieve only 14 of the 16 bytes in the chunk, because the last **Integer** value in the chunk is equivalent to **WAVEFORMATEX.lExtra**, which is always zero in standard PCM files.

```
Private Type WaveFormat
   wFormatTag As Integer
   nChannels As Integer
   nSamplesPerSec As Long
   nAvgBytesPerSec As Long
   nBlockAlign As Integer
   wBitsPerSample As Integer
End Type
```

The third type can be used to retrieve the header of any chunk, including the data chunk.

```
Private Type ChunkHeader
   lType As Long
   lLen As Long
End Type
```

The following sample function verifies that a file is a RIFF wave file, seeks the beginning of the sample data, and returns a **WAVEFORMATEX** type containing information about the wave format:

```
Dim FileFree As Long          ' Global file handle.
Dim lDataLength As Long       ' Global data length.

Private Function FillFormat(FileName As String) As WAVEFORMATEX
```

```
Dim Header As FileHeader
Dim HdrFormat As WaveFormat
Dim chunk As ChunkHeader
Dim by As Byte
Dim i As Long

' Open the file and read the header.

Close #FileFree
FileFree = FreeFile
Open FileName For Binary Access Read As #FileFree
Get #FileFree, , Header

' Check for "RIFF" tag and exit if not found.

If Header.lRiff <> &H46464952 Then
   Exit Function
End If

' Check for "WAVE" tag and exit if not found.

If Header.lWave <> &H45564157 Then
   Exit Function
End If

' Check format chunk length; if less than 16,
' it's not PCM data so we can't use it.

If Header.lFormatLength < 16 Then
   Exit Function
End If

' Retrieve format.

Get #FileFree, , HdrFormat

' Seek to next chunk by discarding any format bytes.

For i = 1 To Header.lFormatLength - 16
   Get #FileFree, , by
Next

' Ignore chunks until we get to the "data" chunk.

Get #FileFree, , chunk
Do While chunk.lType <> &H61746164
```

```
    For i = 1 To chunk.lLen
       Get #FileFree, , by
    Next
    Get #FileFree, , chunk
  Loop

  ' Retrieve the size of the data.

  lDataLength = chunk.lLen

 ' Fill the returned type with the format information.

  With FillFormat
    .lAvgBytesPerSec = HdrFormat.nAvgBytesPerSec
    .lExtra = 0
    .lSamplesPerSec = HdrFormat.nSamplesPerSec
    .nBitsPerSample = HdrFormat.wBitsPerSample
    .nBlockAlign = HdrFormat.nBlockAlign
    .nChannels = HdrFormat.nChannels
    .nFormatTag = HdrFormat.wFormatTag
  End With

End Function
```

The application can now begin reading data from the file and streaming that data into the secondary sound buffer. It's impossible to read data directly from the file into the secondary buffer, so you must first read the data into a private buffer and then copy it to the secondary buffer by using **DirectSoundSecondaryBuffer8.WriteBuffer**.

For more information on streaming, see Using Streaming Buffers and Play Buffer Notification.

Reading Wave Data from a Resource

[Visual Basic]
Because sounds stored as resources are likely to be short, it is recommended that they be played through static buffers. To create a static buffer and fill it with data from a resource, use the **DirectSound8.CreateSoundBufferFromResource** method. For an example, see Filling and Playing Static Buffers.

Longer sounds can be stored in wave files and played through streaming buffers. For more information, see the following topics:

- Using Streaming Buffers
- Reading Wave Files

[C++]
The DirectSound API does not include methods for reading a wave from a resource. However, the Dsutil.cpp file used by many of the SDK sample applications implements a **CWaveFile** class that can read waves stored as resources as well as wave files.

To store wave sounds in an executable, import your wave files as resources and give them string names. Note that **CWaveFile** expects these resources to be of type "WAVE" or "WAV", and expects to find them in the executable module rather than a DLL.

The following code opens a resource identified as "mywave":

```
CWaveFile waveFile;

if (FAILED(waveFile.Open("mywave", NULL, WAVEFILE_READ)))
{
    waveFile.Close();
}
```

Once the resource has been opened, it can be read from just as if it were a file. For more information on how to do this by using the **CWaveFile** class, see Reading Wave Files.

Using Effects in DirectSound

Playback effects in a pure DirectSound application are handled in fundamentally the same way as in an application that uses the DirectMusic performance and audiopaths. For more information, see Using Effects. Because there are no audiopaths in a DirectSound application, you always work directly with buffers.

[C++]
To use effects, a DirectSound application must call **CoInitialize** to initialize COM. Doing so does not preclude creating the DirectSound object by using **DirectSoundCreate8**.

Effects might not work smoothly on very small buffers. DirectSound does not permit the creation of effects-capable buffers that hold less than DSBSIZE_FX_MIN milliseconds of data.

Create a secondary buffer that is capable of holding a DMO by setting the DSBCAPS_CTRLFX flag in the **DSBUFFERDESC** structure.

Implement effects on your secondary sound buffers by using **IDirectSoundBuffer8::SetFX**.

Obtain interfaces for effects by using **IDirectSoundBuffer8::GetObjectInPath**. Note that this method, unlike **IDirectMusicAudioPath8::GetObjectInPath** and

IDirectMusicSegmentState8::GetObjectInPath, does not take a *dwStage* parameter, because DMOs are the only objects that can be retrieved from buffers.

[Visual Basic]
Effects might not work smoothly on very small buffers.DirectSound does not permit the creation of effects-capable buffers that hold less than 150 milliseconds of data.

Create a secondary buffer that is capable of holding a DMO by setting the DSBCAPS_CTRLFX flag in the **DSBUFFERDESC** type.

Implement effects on your secondary sound buffers by using **DirectSoundSecondaryBuffer8.SetFX**.

Obtain objects for effects by using **DirectSoundSecondaryBuffer8.GetObjectInPath**. Note that this method, unlike **DirectMusicAudioPath8.GetObjectInPath**, does not take an *lStage* parameter, because DMOs are the only objects that can be retrieved from buffers.

Sound Capture

This section is a guide to capturing sounds using DirectMusic and DirectSoundCapture objects. The following topics are discussed:

- Capturing MIDI
- Capturing Waves

[Visual Basic]

Note
DirectX for Visual Basic does not support MIDI capture.

Capturing MIDI

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not support MIDI capture.

[C++]
To capture MIDI messages from a device such as a keyboard, create a port for the capture device and use its **IDirectMusicPort8::SetReadNotificationHandle** method to cause an event to be signaled whenever messages are available to be read. In response to the event, call the **IDirectMusicPort8::Read** method repeatedly to place

pending events into a buffer, until S_FALSE is returned. Each time **Read** is called, as many events are put into the buffer as are available, or as fit into the buffer. If at least one event was put into the buffer, S_OK is returned.

To retrieve events from the buffer, call the **IDirectMusicBuffer8::GetNextEvent** method. Each call retrieves a single event, until no more are available, at which point S_FALSE is returned.

The following code fragment illustrates this process. Assume that *hEvent* was created with **CreateEvent** and given to the capture port *pPort* by a call to **SetReadNotificationHandle**. Assume also that *pBuffer* was initialized by **IDirectMusic8::CreateMusicBuffer**.

```
REFERENCE_TIME rt;
DWORD        dwGroup;
DWORD        cb;
BYTE        *pb;

DWORD dw = WaitForMultipleObjects(1, hEvent, FALSE, INFINITE);
for (;;)
{
   hr = pPort->Read(pBuffer);
   if (hr == S_FALSE)
    {
       break;  // No more messages to read into the buffer.
    }
   pBuffer->ResetReadPtr();
   for (;;)
   {
      hr = pBuffer->GetNextEvent(&rt, &dwGroup, &cb, &pb);
      if (hr == S_OK)
      {
         // pb points to the data structure for the message, and
         // you can do anything that you want with it.
      }
      else if (hr == S_FALSE)
      {
         break;  // No more messages in the buffer.
      }
   }  // Done with the buffer.
}  // Done reading pending events.
```

If you don't wish to intercept messages, but simply want to send them from one port to another, you can use the **IDirectMusicThru8** interface. See **IDirectMusicThru8::ThruChannel** for details.

Capturing Waves

[C++]
The DirectSoundCapture object (**IDirectSoundCapture8**) is used to query the capabilities of sound capture devices and to create buffers for capturing audio from an input source. DirectSoundCapture allows capturing of data in PCM or compressed formats.

The DirectSound capture buffer object (**IDirectSoundCaptureBuffer8**) represents a buffer used for receiving data from the input device. Like playback buffers, this buffer is conceptually circular: when input reaches the end of the buffer, it automatically starts again at the beginning.

The methods of **IDirectSoundCaptureBuffer8** enable you to retrieve the properties of the buffer, start and stop audio capture, and lock portions of the memory, so that you can safely retrieve data for saving to a file or for some other purpose. On operating systems that support capture effects, it also provides methods for retrieving DMOs and ascertaining the status of effects.

[Visual Basic]
The **DirectSoundCapture8** object is used to query the capabilities of sound capture devices and to create buffers for capturing audio from an input source. **DirectSoundCapture8** allows capturing of data in PCM or compressed formats.

The **DirectSoundCaptureBuffer8** object represents a buffer used for receiving data from the input device. Like playback buffers, this buffer is conceptually circular: when input reaches the end of the buffer, it automatically starts again at the beginning.

The methods of the **DirectSoundCaptureBuffer8** object allow you to retrieve the properties of the buffer, start and stop audio capture, and lock portions of the memory, so that you can safely retrieve data for saving to a file or for some other purpose.

More information about wave capture is found in the following topics:

- Enumeration of Capture Devices
- Creating the DirectSoundCapture Object
- Capture Device Capabilities
- Creating a Capture Buffer
- Capture Buffer Information
- Capture Buffer Notification
- Capture Buffer Effects
- Using the Capture Buffer

- Writing to a Wave File

Enumeration of Capture Devices

[C++]
For an application that is simply going to capture sounds through the user's preferred capture device, you don't need to enumerate the available devices. By passing NULL or a predefined GUID to **DirectSoundCaptureCreate8** or **DirectSoundFullDuplexCreate8**,you can create a default device. For more information, see Creating the DirectSoundCapture Object.

[Visual Basic]
For an application that is simply going to capture sounds through the user's preferred capture device, you don't need to enumerate the available devices. By passing **vbNullString** or a predefined GUID to **DirectX8.DirectSoundCaptureCreate**, you can create a default device. For more information, see Creating the DirectSoundCapture Object.

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

Enumeration serves three purposes:

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

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

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

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

```
DWORD pv;  // Can be any 32-bit type.

HRESULT hr = DirectSoundCaptureEnumerate(
```

```
(LPDSENUMCALLBACK)DSEnumProc,
(VOID*)&pv);
```

[Visual Basic]
To enumerate devices, first call the **DirectX8.GetDSCaptureEnum** method to create a **DirectSoundEnum8** object representing a collection of devices. You can then query each device by using the **DirectSoundEnum8.GetDescription**, **DirectSoundEnum8.GetGuid**, and **DirectSoundEnum8.GetName** methods.

For an example of the use of **DirectSoundEnum8**, see Enumeration of Sound Devices.

Creating the DirectSoundCapture Object

[C++]
Create the DirectSoundCapture object by calling the **DirectSoundCaptureCreate8** or **DirectSoundFullDuplexCreate8** function. Both these functions retrieve a pointer to the **IDirectSoundCapture8** interface.

The *lpcGUID* parameter to **DirectSoundCaptureCreate**, or the *pcGuidRenderDevice* parameter to **DirectSoundFullDuplexCreate8**, can be a GUID obtained by enumeration, or it can be one of the following predefined GUIDs:

DSDEVID_DefaultCapture
: The default system capture device. You can also specify this device by passing a NULL pointer in the device GUID parameter.

DSDEVID_DefaultVoiceCapture
: The default voice communications capture device. Typically, this is a secondary device such as a USB headset with microphone.

If no device driver is present, the call fails.

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

If you want DirectSound and DirectSoundCapture objects to coexist, then you should create and initialize the DirectSound object before creating and initializing the DirectSoundCapture object, or use **DirectSoundFullDuplexCreate8**.

Some audio devices aren't configured for full duplex audio by default. If your application has problems with creating and initializing both a DirectSound object and a DirectSoundCapture object, you should advise the user to check the audio device properties to ensure that full duplex is enabled.

[Visual Basic]
You create the **DirectSoundCapture8** object by calling the **DirectX8.DirectSoundCaptureCreate** method.

The *guid* parameter to **DirectSoundCaptureCreate** can be a GUID obtained by enumeration, or it can be one of the following values:

DSDEVID_DefaultCapture
: The default system capture device. You can also specify this device by passing **vbNullString** in the device GUID parameter.

DSDEVID_DefaultVoiceCapture
: The default voice communications capture device. Typically, this is a secondary device such as a USB headset with microphone.

If no device driver is present, the call raises an error.

If you want **DirectSound8** and **DirectSoundCapture8** objects to coexist, then you should create the **DirectSound8** object before creating the **DirectSoundCapture8** object.

Some audio devices aren't configured for full duplex audio by default. If your application has problems creating both a **DirectSound8** object and a **DirectSoundCapture8** object, you should advise the user to check the audio device properties to ensure that full duplex is enabled.

Capture Device Capabilities

[C++]
To retrieve the capabilities of a capture device, call the **IDirectSoundCapture8::GetCaps** method. The parameter to this method is a pointer to a **DSCCAPS** structure. As with other such structures, you have to initialize the **dwSize** member before passing it. On return, the structure contains the number of channels the device supports, as well as a combination of values for supported formats, equivalent to the values in the **WAVEINCAPS** structure used in the Win32 waveform audio functions.

[Visual Basic]
To retrieve the capabilities of a capture device, call the **DirectSoundCapture8.GetCaps** method. The parameter to this method is a **DSCCAPS** type. On return, the type contains the number of channels the device supports as well as a combination of values for supported formats.

On non-WDM drivers, a capture device can be used by only one application at a time. If the driver supports simultaneous use of the device by more than one application,

DSCCAPS_MULTIPLECAPTURE is returned in the **dwFlags** member of **DSCCAPS**. Each application can set its own format for the capture buffer.

Creating a Capture Buffer

[C++]
Create a capture buffer by calling the **IDirectSoundCapture8::CreateCaptureBuffer** method.

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

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

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

```
// pDSC is a valid IDirectSoundCapture8 interface pointer.

DSCBUFFERDESC            dscbd;
LPDIRECTSOUNDCAPTUREBUFFER pDSCB;
WAVEFORMATEX             wfx =
   {WAVE_FORMAT_PCM, 2, 44100, 176400, 4, 16, 0};

   // wFormatTag, nChannels, nSamplesPerSec, mAvgBytesPerSec,
   // nBlockAlign, wBitsPerSample, cbSize

dscbd.dwSize = sizeof(DSCBUFFERDESC);
dscbd.dwFlags = 0;
dscbd.dwBufferBytes = wfx.nAvgBytesPerSec;
dscbd.dwReserved = 0;
dscbd.lpwfxFormat = &wfx;
dscbd.dwFXCount = 0;
dscbd.lpDSCFXDesc = NULL;

pDSCB = NULL;

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

[Visual Basic]

Create a capture buffer by calling the **DirectSoundCapture8.CreateCaptureBuffer** method.

One of the parameters to the method is a **DSCBUFFERDESC** type that describes the characteristics of the desired buffer. The **fxFormat** member is a **WAVEFORMATEX** type, which must be initialized with the details of the desired wave format.

Note that if your application is using **DirectSound8** as well as **DirectSoundCapture8**, capture buffer creation can fail when the format of the capture buffer is not the same as that of the primary buffer. The reason is that some cards have only a single clock and cannot support capture and playback at two different frequencies.

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

```
' dsc is a DirectSoundCapture8 object.

Dim dscbd As DSCBUFFERDESC
Dim dscb As DirectSoundCaptureBuffer8

' Set up the wave format.
With dscbd.fxFormat
    .nFormatTag = WAVE_FORMAT_PCM
    .nChannels = 2
    .lSamplesPerSec = 22050
    .nBitsPerSample = 16
    .nBlockAlign = _
    .nBitsPerSample / 8 * WaveFormat.nChannels
    .lAvgBytesPerSec = _
          waveFormat.lSamplesPerSec * waveFormat.nBlockAlign
    .nSize = 0    ' Ignored for WAVE_FORMAT_PCM.
End With

dscbd.lFlags = DSCBCAPS_DEFAULT
dscbd.lBufferBytes = dscbd.fxFormat.lAvgBytesPerSec

Set dscb = dsc.CreateCaptureBuffer (dscbd)
```

Capture Buffer Information

[C++]
Use the **IDirectSoundCaptureBuffer8::GetCaps** method to retrieve the size of a capture buffer. Be sure to initialize the **dwSize** member of the **DSCBCAPS** structure before passing it as a parameter.

You can also retrieve information about the format of the data in the buffer, as set when the buffer was created. Call the **IDirectSoundCaptureBuffer8::GetFormat** method, which returns the format information in a **WAVEFORMATEX** structure.

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

To find out what a capture buffer is currently doing, call the **IDirectSoundCaptureBuffer8::GetStatus** method. This method fills a **DWORD** variable with a combination of flags that indicate whether the buffer is busy capturing, and if so, whether it is looping; that is, whether the DSCBSTART_LOOPING flag was set in the last call to **IDirectSoundCaptureBuffer8::Start**.

Finally, the **IDirectSoundCaptureBuffer8::GetCurrentPosition** method returns the offsets of the read and capture cursors within the buffer. The read cursor is at the end of the data that has been fully captured into the buffer at this point. The capture cursor is at the end of the block of data that is currently being copied from the hardware. You can safely copy data from the buffer only up to the read cursor.

[Visual Basic]
Use the **DirectSoundCaptureBuffer8.GetCaps** method to retrieve information about buffer capabilities. To get information about the data format of the buffer, set when the buffer was created, call the **DirectSoundCaptureBuffer8.GetFormat** method.

To find out what a capture buffer is doing, call the **DirectSoundCaptureBuffer8.GetStatus** method. This method returns with a combination of flags that indicate whether the buffer is busy capturing, and if so, whether it is looping; that is, whether the DSCBSTART_LOOPING flag was set in the last call to **DirectSoundCaptureBuffer8.Start**.

Finally, the **DirectSoundCaptureBuffer8.GetCurrentPosition** method returns the offsets of the read and capture cursors within the buffer. The read cursor is at the end of the data that has been fully captured into the buffer at this point. The capture cursor is at the end of the block of data that is currently being copied from the hardware. You can safely copy data from the buffer only up to the read cursor.

Capture Buffer Notification

[C++]
You may want your application to be notified when the read cursor reaches a certain point in the buffer, or when it reaches the end. The read cursor is at the point up to which it is safe to read data from the buffer. With the

IDirectSoundNotify8::SetNotificationPositions method, you can set any number of points within the buffer where events are to be signaled.

First, obtain a pointer to the **IDirectSoundNotify8** interface. You can do this with the capture buffer's **QueryInterface** method, as shown in the example under Play Buffer Notification.

Next create an event object with the Win32 **CreateEvent** function. You put the handle to this event in the **hEventNotify** member of a **DSBPOSITIONNOTIFY** structure, and in the **dwOffset** member of that structure, you specify the offset within the buffer where you want the event to be signaled. Then, you pass the address of the structure—or array of structures, if you want to set more than one notification position—to the **IDirectSoundNotify8::SetNotificationPositions** method.

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

```
#define cEvents  3

// It is assumed that the following variables have
// been properly initialized, and that wfx was included in the
// buffer description when the buffer was created.
//
//   LPDIRECTSOUNDNOTIFY8 lpDsNotify;
//   WAVEFORMATEX         wfx;

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

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

// Describe notifications.

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

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

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

// Create notifications.

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

[Visual Basic]
You may want your application to be notified when the read cursor reaches a certain point in the buffer, or when it reaches the end. The read cursor is at the point up to which it is safe to read data from the buffer. With the **DirectSoundCaptureBuffer8.SetNotificationPositions** method, you can set any number of points within the buffer where events are to be signaled.

To set notification events on a buffer, you must first implement the **DirectXEvent8** class in the form that contains the code for the buffer. Create a **DirectXEvent8.DXCallback** method by choosing **DirectXEvent8** from the drop-down class list, and add the code you want to be executed when the event is signaled.

Create an event handle by calling the **DirectX8.CreateEvent** method and pass the event handle in the **hEventNotify** member of a **DSBPOSITIONNOTIFY** type. Pass the type to the **DirectSoundCaptureBuffer8.SetNotificationPositions** method. The event handle will now be passed to **DirectXEvent8.DXCallback** whenever the play cursor reaches that position.

Capture Buffer Effects

[C++]
Capture buffer effects are not available on current operating systems.

Set effects on capture buffers by describing them in the **DSCBUFFERDESC** structure passed to **IDirectSoundCapture8::CreateCaptureBuffer**. After the buffer has been created, retrieve information about the effects by using **IDirectSoundCaptureBuffer8::GetFXStatus**.

The following interfaces represent capture effects supported by DirectX 8.0, and can be used to set and retrieve parameters:

- **IDirectSoundCaptureFXAec8**
- **IDirectSoundCaptureFXNoiseSuppress8**

These interfaces are obtained from the capture buffer object by using **IDirectSoundCaptureBuffer8::GetObjectInPath**.

[Visual Basic]
Information in this topic pertains only to applications written in C++. DirectX for Visual Basic does not support capture buffer effects.

Using the Capture Buffer

[C++]
Capturing a sound consists of the following steps:

1. Start the buffer by calling the **IDirectSoundCaptureBuffer8::Start** method. Audio data from the input device begins filling the buffer from the beginning.
2. Wait until the desired amount of data is available. For one method of determining when the capture cursor reaches a certain point, see Capture Buffer Notification.
3. When sufficient data is available, lock a portion of the capture buffer by calling the **IDirectSoundCaptureBuffer8::Lock** method.

   To make sure you are not attempting to lock a portion of memory that is about to be used for capture, you can first obtain the position of the read cursor by calling **IDirectSoundCaptureBuffer8::GetCurrentPosition**. For an explanation of the read cursor, see Capture Buffer Information.

   As parameters to the **Lock** method, you pass the size and offset of the block of memory you want to read. The method returns a pointer to the address where the memory block begins, and the size of the block. If the block wraps around from the end of the buffer to the beginning, two pointers are returned, one for each section of the block. The second pointer is NULL if the locked portion of the buffer does not wrap around.
4. Copy the data from the buffer, using the addresses and block sizes returned by the **Lock** method.
5. Unlock the buffer with the **IDirectSoundCaptureBuffer8::Unlock** method.
6. Repeat steps 2 to 5 until you are ready to stop capturing data. Then call the **IDirectSoundCaptureBuffer8::Stop** method.

Normally, the buffer stops capturing automatically when the capture cursor reaches the end of the buffer. However, if the DSCBSTART_LOOPING flag was set in the *dwFlags* parameter to the **IDirectSoundCaptureBuffer8::Start** method, the capture will continue until the application calls the **IDirectSoundCaptureBuffer8::Stop** method.

[Visual Basic]

Capturing a sound consists of the following steps:

1. Start the buffer by calling the **DirectSoundCaptureBuffer8.Start** method. Audio data from the input device begins filling the buffer from the beginning.
2. Wait until the desired amount of data is available. For one method of determining when the capture cursor reaches a certain point, see Capture Buffer Notification.
3. When sufficient data is available, read a portion of the capture buffer by calling the **DirectSoundCaptureBuffer8.ReadBuffer** method.

   To make sure you are not attempting to read a portion of memory that is about to be used for capture, you can first obtain the position of the read cursor by calling **DirectSoundCaptureBuffer8.GetCurrentPosition**. For an explanation of the read cursor, see Capture Buffer Information.
4. Repeat steps 2 and 3 until you are ready to stop capturing data. Then call the **DirectSoundCaptureBuffer8.Stop** method.

By default, the buffer stops capturing automatically when the capture cursor reaches the end of the buffer. However, if the DSCBSTART_LOOPING flag was set in the *flags* parameter to the **DirectSoundCaptureBuffer8.Start** method, the capture continues until the application calls the **DirectSoundCaptureBuffer8.Stop** method.

The following procedure in a form that implements **DirectXEvent8** is called each time the capture cursor reaches a notification position. It calculates the amount of new data available, copies it to an intermediate buffer, and then writes it to a file.

```
Private Sub DirectXEvent8_DXCallback(ByVal eventid As Long)

' Global variables:
' lastPos As Long – initialized to 0 each time capture starts
' BytesWritten As Long – also initialized to 0
' dscb As DirectSoundCaptureBuffer8
' dscbDesc As DSCBUFFERDESC
' EventStop As Long

  Static curPos As Long
  Static curs As DSCURSORS
  Dim dataBuf() As Byte
  Dim dataSize As Long

  dscb.GetCurrentPosition curs
  curPos = curs.lWrite  ' Position up to which data is valid

  ' Create an intermediate, temporary data buffer.
  ' lastPos is a global Long that is initialized to 0 each time
  ' capture begins.

  dataSize = curPos - lastPos
  If dataSize < 0 Then        ' curPos wrapped around.
```

```
  dataSize = dscbDesc.lBufferBytes - lastPos + curPos
End If

ReDim dataBuf(dataSize - 1)
dscb.ReadBuffer lastPos, dataSize, dataBuf(0), DSCBLOCK_DEFAULT

Put #1, , dataBuf
BytesWritten = BytesWritten + dataSize
lastPos = curPos

If (eventid = EventStop) Then
  CloseFile
End If

End Sub
```

The *EventStop* event was associated with the offset DSBPN_OFFSETSTOP when notification positions were set. This event is signaled when the capture buffer stops. In response, the application calls a procedure that adds information to the file header and closes the file. For information on how to write file headers for wave files, see Writing to a Wave File.

Writing to a Wave File

[C++]
The DirectSound API does not include methods for writing to wave files. However, the Dsutil.cpp file used by many of the SDK sample applications implements a **CWaveFile** class that has the following public methods:

- **Open**. Opens a file for writing and retrieves the wave format, or opens the file for writing and writes the header chunks.
- **GetSize**. Returns the size of the data chunk, after **Open** or **ResetFile** has been called.
- **Read**. Copies a portion of the data chunk into a buffer and advances the read cursor.
- **Write**. Writes from a buffer to the data chunk and advances the write cursor.
- **ResetFile**. Sets the read and write cursors to the beginning of the data chunk.
- **Close**. Closes the file.

The first step in writing a wave file is to call the **CWaveFile::Open** method. This creates the file and writes the wave format chunk. The parameters are the filename, a pointer to an initialized **WAVEFORMATEX** structure, and the WAVEFILE_WRITE flag. The method returns an **HRESULT**.

The following code opens a wave file for writing:

```
CWaveFile     g_pWaveFile;
WAVEFORMATEX  wfxInput;

ZeroMemory( &wfxInput, sizeof(wfxInput));
wfxInput.wFormatTag = WAVE_FORMAT_PCM;
wfxInput.nSamplesPerSec = 22050
wfxInput.wBitsPerSample =  8;
wfxInput.nChannels = 1;
wfxInput.nBlockAlign =
   wfxInput.nChannels * (wfxInput.wBitsPerSample / 8);
wfxInput.nAvgBytesPerSec =
   wfxInput.nBlockAlign * wfxInput.nSamplesPerSec;

g_pWaveFile = new CWaveFile;
if (FAILED(g_pWaveFile->Open("mywave.wav", &wfxInput,
      WAVEFILE_WRITE)))
{
   g_pWaveFile->Close();
}
```

The application can now begin copying data from the capture buffer to the file. The following function is called each time the read cursor reaches a notification position. The following global variables are used:

- *g_pDSBCapture* is a pointer to the **IDirectSoundCapture** interface of the capture buffer.
- *g_dwNextCaptureOffset* tracks the buffer offset of the next block of data that will be copied to the file.
- *g_dwCaptureBufferSize* is the size of the capture buffer, used in calculating wraparound.

```
HRESULT RecordCapturedData()
{
   HRESULT hr;
   VOID*   pbCaptureData    = NULL;
   DWORD   dwCaptureLength;
   VOID*   pbCaptureData2   = NULL;
   DWORD   dwCaptureLength2;
   VOID*   pbPlayData       = NULL;
   UINT    dwDataWrote;
   DWORD   dwReadPos;
   LONG lLockSize;

   if (NULL == g_pDSBCapture)
```

```
    return S_FALSE;
if (NULL == g_pWaveFile)
    return S_FALSE;

if (FAILED( hr = g_pDSBCapture->GetCurrentPosition(
        NULL, &dwReadPos)))
    return hr;

// Lock everything between our private cursor
// and the read cursor, allowing for wraparound.

lLockSize = dwReadPos - g_dwNextCaptureOffset;
if( lLockSize < 0 )
    lLockSize += g_dwCaptureBufferSize;

if( lLockSize == 0 )
    return S_FALSE;

if (FAILED(hr = g_pDSBCapture->Lock(
        g_dwNextCaptureOffset, lLockSize,
        &pbCaptureData, &dwCaptureLength,
        &pbCaptureData2, &dwCaptureLength2, 0L)))
return hr;

// Write the data. This is done in two steps
// to account for wraparound.

if (FAILED( hr = g_pWaveFile->Write( dwCaptureLength,
        (BYTE*)pbCaptureData, &dwDataWrote)))
    return hr;

if (pbCaptureData2 != NULL)
{
    if (FAILED(hr = g_pWaveFile->Write(
            dwCaptureLength2, (BYTE*)pbCaptureData2,
            &dwDataWrote)))
            return hr;
}

// Unlock the capture buffer.

g_pDSBCapture->Unlock( pbCaptureData, dwCaptureLength,
        pbCaptureData2, dwCaptureLength2  );

// Move the capture offset along.
```

```
    g_dwNextCaptureOffset += dwCaptureLength;
    g_dwNextCaptureOffset %= g_dwCaptureBufferSize;
    g_dwNextCaptureOffset += dwCaptureLength2;
    g_dwNextCaptureOffset %= g_dwCaptureBufferSize;

    return S_OK;
}
```

When capturing is finished, the application closes the wave file.

```
g_pWaveFile->Close();
```

The **CWaveFile::Close** method writes the chunk sizes to the file before closing it.

[Visual Basic]
Short sounds can be saved from a secondary buffer to a wave file by using **DirectSoundSecondaryBuffer8.SaveToFile**. This method copies the entire contents of the buffer to a file.

In most cases, the sound that you want to save starts out in a capture buffer. The following code shows how to move data from a capture buffer, *dscb*, to a secondary buffer and then to a file.

```
Dim ds As DirectSound8
Dim dscb As DirectSoundCaptureBuffer8
Dim dscd As DSCBUFFERDESC
Dim dsd As DSBUFFERDESC
Dim capCurs As DSCURSORS
Dim numBytes As Long
Dim ByteBuffer() As Integer
.
.
.
' Assume that ds has been created, dscb has been created and
' contains data, and dscd holds the format of the capture buffer.

' Create a secondary buffer with the same format, and large enough
' to contain all data so far written to the capture buffer.

dscb.GetCurrentPosition capCurs
numBytes = capCurs.lWrite + 1
dsd.lBufferBytes = numBytes
dsd.fxFormat = dscd.fxFormat
Set dsb = ds.CreateSoundBuffer(dsd)

' Make a private buffer large enough to hold the same data.
```

```
ReDim ByteBuffer(numBytes)

' Read the contents of the capture buffer into the private buffer.

dscb.ReadBuffer 0, numBytes, ByteBuffer(0), DSCBLOCK_DEFAULT

' Write the private buffer to the secondary buffer.

dsb.WriteBuffer 0, numBytes, ByteBuffer(0), DSBLOCK_DEFAULT

' Save the wave.

dsb.SaveToFile "captured.wav"
```

For sounds that might exceed the length of the capture buffer, you need to create the wave file yourself and stream data into it. Streaming captured data to a wave file is done in the following steps:

1. Create the file and write the file header, the format chunk, and the header of the data chunk. The format chunk must describe the format of the capture buffer. Note, however, that the elements of the format are not in the same order as in the **WAVEFORMATEX** type. For information on the organization of headers and chunks, see Reading Wave Files.
2. Stream data from a capture buffer to a private buffer and from there to the file, keeping track of how many bytes have been written. For more information, see Using the Capture Buffer.
3. When capture stops and all data has been written, calculate the length of the data chunk and the total size of the file, and write these values to the appropriate locations in the data chunk header and file header respectively.

The following code creates a file and writes the RIFF header information:

```
Private Type FileHeader
    lRiff As Long
    lFileSize As Long
    lWave As Long
    lFormat As Long
    lFormatLength As Long
End Type

Private Type WaveFormat
    wFormatTag As Integer
    nChannels As Integer
    nSamplesPerSec As Long
    nAvgBytesPerSec As Long
    nBlockAlign As Integer
```

```
    wBitsPerSample As Integer
'   cbSize As Integer
End Type

Private Type ChunkHeader
    lType As Long
    lLen As Long
End Type

Dim fh As FileHeader
Dim wf As WaveFormat
Dim ch As ChunkHeader

Private Sub OpenFile(WaveFileName As String)

  Open WaveFileName For Binary Access Write As #1

  With fh
    .lRiff = &H46464952    ' <RIFF> chunk tag
    .lFileSize = 0         ' Will get later
    .lWave = &H45564157    ' <WAVE> chunk tag
    .lFormat = &H20746D66  ' <fmt > chunk tag
    .lFormatLength = Len(wf)
  End With

  Put #1, , fh

  With wf
    .wFormatTag = dscbDesc.fxFormat.nFormatTag
    .nChannels = dscbDesc.fxFormat.nChannels
    .nSamplesPerSec = dscbDesc.fxFormat.lSamplesPerSec
    .wBitsPerSample = dscbDesc.fxFormat.nBitsPerSample
    .nBlockAlign = dscbDesc.fxFormat.nBlockAlign
    .nAvgBytesPerSec = dscbDesc.fxFormat.lAvgBytesPerSec
  End With

  Put #1, , wf

    ch.lType = &H61746164  ' <data> chunk tag

  Put #1, , ch

End Sub
```

Captured data can now be appended to the file. Assuming that the number of data bytes written to the file is kept in a **Long** called *BytesWritten*, the file is closed by using the following procedure:

```
Private Sub CloseFile()

  Dim fsize As Long

  ' Write file size.

  fsize = Len(fh) + Len(wf) + Len(ch) + BytesWritten
  Put #1, 5, fsize

  ' Rewrite data chunk header with size.

  ch.lLen = BytesWritten
  Put #1, Len(fh) + Len(wf) + 1, ch

  Close #1

End Sub
```

Advanced Topics in DirectX Audio

This section contains information needed for specialized applications that need functionality beyond that covered under Using DirectX Audio.

Information is presented in the following topics:

- DirectMusic Tracks
- Using DirectMusic Messages
- Using DirectMusic Ports
- Custom Loading
- Using Instrument Collections
- Low-Level DLS
- DirectMusic Tools
- Property Sets
- Optimizing DirectSound Performance
- Writing to the Primary Buffer

[Visual Basic]
Many of the topics in this section do not pertain to development with Microsoft® Visual Basic®, because the Microsoft® DirectX® for Visual Basic type library does not support some of the more advanced features of the C++ API. These topics appear in the table of contents because the C++ and Visual Basic versions of DirectX Help are based on common source material.

DirectMusic Tracks

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not provide applications with access to individual tracks.

[C++]
Tracks are the components of a segment that contain its sequenced data, including information about notes, underlying chords, tempo, patch and band changes, and everything else that the performance needs to know to play the sounds.

Each track is represented by an **IDirectMusicTrack8** interface. The methods of this interface are called by the performance, and most applications don't need to use them directly. This interface is chiefly of interest for plug-in components that implement their own track types.

When an application calls **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx**, DirectMusic calls the **IDirectMusicTrack8::Play** or **IDirectMusicTrack8::PlayEx** method on the segment's tracks. Most tracks respond by immediately generating time-stamped messages containing data that is valid for the part of the segment that is being played. These messages are placed in a queue. See Message Creation and Delivery for more information about what happens after that.

A few tracks do not actively generate messages other than notifications in response to **IDirectMusicTrack8::Play** or **IDirectMusicTrack8::PlayEx**, but instead do most of their work by responding to requests for information that come from the performance or other tracks. The most important of these are the chord, mute, and command tracks. The tempo track sends messages but also responds to parameter requests.

More information is contained in the following topics:

- Standard Track Types
- Track Configuration

Standard Track Types

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not provide applications with access to individual tracks.

[C++]
The following list describes the standard track types implemented by Microsoft® DirectMusic®. The class identifiers, such as CLSID_DirectMusicBandTrack, are used to identify track types in calls to various methods.

Track type	Class GUID	Purpose
Band	CLSID_DirectMusicBandTrack	Downloads DLS data to the performance. Sends messages of type **DMUS_PATCH_PMSG**, **DMUS_TRANSPOSE_PMSG**, **DMUS_CHANNEL_PRIORITY_PMSG**, and **DMUS_MIDI_PMSG** (for volume and pan). Used in segments based on MIDI files and styles.
Chord	CLSID_DirectMusicChordTrack	Used to convert music values in patterns to MIDI values. Sends messages of type **DMUS_NOTIFICATION_PMSG** (for GUID_NOTIFICATION_CHORD notifications).
Chordmap	CLSID_DirectMusicChordMapTrack	Used in template segments to compose chord tracks.
Command	CLSID_DirectMusicCommandTrack	Used in template segments to compose chord tracks, and in style segments to determine which patterns are played. Sends messages of type **DMUS_NOTIFICATION_PMSG** for GUID_NOTIFICATION_COMMAND notifications.
Lyrics	CLSID_DirectMusicLyricsTrack	Used to synchronize words with music. Generates messages of type **DMUS_LYRIC_PMSG**.
Marker	CLSID_DirectMusicMarkerTrack	Used for flow control. The marker track can hold valid start times for the segment, and play times at which new segments can be cued. For more information, see Segment Timing.
Motif	CLSID_DirectMusicMotifTrack	Used to play motifs to accompany other segments. Sends messages of type **DMUS_CURVE_PMSG**, **DMUS_NOTE_PMSG**, and **DMUS_NOTIFICATION_PMSG** (for GUID_NOTIFICATION_MEASUREANDBEAT notifications).
Mute	CLSID_DirectMusicMuteTrack	Enables performance channels to be remapped or muted. Used with either style-based or MIDI-

		based segments.
Parameter control	CLSID_DirectMusicParamControlTrack	Controls the settings on tools, effects, and any other objects that support the **IMediaParams** interface.
Pattern	CLSID_DirectMusicPatternTrack	Contains a single musical pattern. The pattern track is similar to a sequence track, but contains music values rather than fixed notes. This track makes it possible to audition a pattern against different chords, and is used mostly by music-authoring applications. It might also be used to play an accompaniment. It is represented by its own interface, **IDirectMusicPatternTrack8**.
Script	CLSID_DirectMusicScriptTrack	Calls routines in an audio script.
Segment trigger	CLSID_DirectMusicSegmentTriggerTrack	Triggers the playback of segments. This track enables the author of a file to cue a segment from within a segment, rather than leaving this up to the application developer.
Sequence	CLSID_DirectMusicSeqTrack	Sends sequence messages of type **DMUS_NOTE_PMSG** and **DMUS_MIDI_PMSG**. Used in segments based on MIDI files. Also sends messages of type **DMUS_CURVE_PMSG** for segments saved in the .sgt format.
Signpost	CLSID_DirectMusicSignPostTrack	Used in template segments to compose chord tracks.
Style	CLSID_DirectMusicStyleTrack	Fundamental track for segments based on styles. Sends messages of type **DMUS_TIMESIG_PMSG**, **DMUS_CURVE_PMSG**, **DMUS_NOTE_PMSG**, and **DMUS_NOTIFICATION_PMSG** (for GUID_NOTIFICATION_MEASUREANDBEAT notifications).
SysEx	CLSID_DirectMusicSysExTrack	Sends system exclusive messages of type **DMUS_SYSEX_PMSG**. Used in segments based on MIDI files.
Tempo	CLSID_DirectMusicTempoTrack	Controls the tempo of the performance.
Time Signature	CLSID_DirectMusicTimeSigTrack	Sends messages of type **DMUS_TIMESIG_PMSG** as well as GUID_NOTIFICATION_MEASUREANDBEAT notifications. The time signature track exists in imported MIDI files and authored segments specifically created with one. In most cases, the

		style track implements the time signature track's functionality, so it is not necessary for a segment that contains a style track to contain a time signature track as well.
Wave	CLSID_DirectMusicWaveTrack	Sends messages of type **DMUS_WAVE_PMSG** to play time-stamped wave sounds.

Track Configuration

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not provide applications with access to individual tracks.

[C++]
Using the **IDirectMusicSegment8::SetTrackConfig** and **IDirectMusicSegmentState8::SetTrackConfig** methods, an application can modify the behavior of any track in a segment. Configuration options include the following:

- Enable or disable playback.
- Enable or disable parameter calls on a track in a control segment.
- Enable or disable notifications.
- Override notifications in a primary segment with notifications from a secondary control segment.
- Enable or disable track composition.
- Control the start point of track data used in composing transitions.

For a list of the track configuration flags and a table of the flags valid on standard tracks, see **IDirectMusicSegmentState8::SetTrackConfig**.

Two of the configuration flags, DMUS_TRACKCONFIG_CONTROL_PLAY and DMUS_TRACKCONFIG_CONTROL_NOTIFICATION, extend the powers of secondary control segments. Normally, a secondary control segment manages only parameters that are obtained by the performance through calls to **IDirectMusicTrack8::GetParam**. Like any segment, the control segment can also make changes to the performance by sending messages. For example, it might change the volume of the performance. Such changes might appear to be overriding parameters in the primary segment, but they differ from true control changes in two ways:

- They are valid only until a similar change is made by another segment. Control segment parameters cannot be overridden by other segments.

- As long as they are not overridden, they remain valid even after the sending segment has finished playing. Control segment parameters are valid only until another segment becomes the control segment.

When the DMUS_TRACKCONFIG_CONTROL_PLAY or DMUS_TRACKCONFIG_CONTROL_NOTIFICATION flag is set on a track, the equivalent track on the primary segment is disabled. It is reenabled when the controlling segment stops.

The **SetTrackConfig** method is available on the **IDirectMusicSegmentState8** interface as well as on **IDirectMusicSegment8**. The parameters are identical, but the behavior differs as follows:

- If you change a flag on a segment, subsequent instances of playing segment states inherit the changed flags. However, segment states that are already playing do not change their behavior.
- If you change a flag on a segment state, the behavior changes immediately for that segment state only.

Note

To ensure that the track configuration on a segment state changes immediately rather than after prepare time, you can call **IDirectMusicPerformance8::Invalidate**.

The following example code disables a chord progression track in the segment addressed by *pSegment*. Chord progressions are broadcast as control segment parameters, so this is done by turning off the DMUS_TRACKCONFIG_CONTROL_ENABLED flag:

```
HRESULT hr = pSegment->SetTrackConfig(CLSID_DirectMusicChordTrack,
    -1, DMUS_SEG_ALLTRACKS, 0, DMUS_TRACKCONFIG_CONTROL_ENABLED);
```

The next example does the opposite, enabling all chord tracks to play:

```
hr = pSegment->SetTrackConfig(CLSID_DirectMusicChordTrack,
    -1, DMUS_SEG_ALLTRACKS, DMUS_TRACKCONFIG_CONTROL_ENABLED, 0);
```

The following topics contain related information:

- Track Composition
- Control Segments
- Self-Controlling Segments

Using DirectMusic Messages

Sound data passes through the DirectMusic performance in the form of messages. For the most part, messages are created and processed behind the scenes, and your application might never have to work directly with them. However, a basic knowledge of messages can help you understand how DirectMusic works, and a more thorough understanding will enable you to use messages for greater control over the performance.

DirectMusic uses two different kinds of messages:

- Performance messages. All sequenced data passes through the performance engine in this form. These messages contain detailed information about timing and routing of the data.
- Standard MIDI messages. These can be read from a MIDI file or device and either passed directly (thrued) to another device or converted to performance message format when played by the performance.

Applications don't deal directly with MIDI messages. When a segment is played, all its data is in the form of performance messages and stays that way until it reaches the final output tool, which converts it to MIDI message format before sequencing it to the synthesizer. However, some performance messages contain information similar to that in standard MIDI messages. To help you understand such messages, this section describes some aspects of the MIDI message format.

[C++]

Note

The **DMUS_MIDI_PMSG** structure contains data equivalent to that in any standard MIDI message. It is, however, a performance message.

Most performance messages are sent by a segment's tracks as the segment is playing. Applications can also send messages to do things like setting a MIDI controller, playing a single note, or changing the tempo.

Once a message has been sent, the application cannot retrieve or alter it except by implementing a tool. For example, a segment authored in DirectMusic Producer might contain a lyrics track that generates **DMUS_LYRIC_PMSG** messages. The only way an application can display the lyrics is by implementing a tool designed for that purpose. For more information, see DirectMusic Tools.

Notifications are an exception to the rule that messages can be intercepted only by tools. The **IDirectMusicPerformance8::GetNotificationPMsg** method enables the application to retrieve **DMUS_NOTIFICATION_PMSG** messages. For more information, see Notification and Event Handling.

[Visual Basic]

Note

The **DirectMusicPerformance8.SendMIDIPMSG** method sends data equivalent to that in any standard MIDI message. However, the data is in the form of a performance message.

Most performance messages are sent by a segment's tracks as the segment is playing. Applications can also send messages to do things like setting a MIDI controller, playing a single note, or changing the tempo.

Once a message has been sent, the application cannot retrieve or alter it. For example, a segment authored in DirectMusic Producer might contain a lyrics track that generates text messages. DirectX for Visual Basic does not enable an application to retrieve these messages and display the lyrics.

Notifications are an exception to the rule that messages cannot be retrieved by the application. The **DirectMusicPerformance8.GetNotificationPMsg** method enables the application to retrieve **DMUS_NOTIFICATION_PMSG** messages. For more information, see Notification and Event Handling.

The following topics discuss messages and how they are routed:

- Channels
- Message Creation and Delivery
- Application-Created Messages
- Performance Message Types
- Curves
- MIDI Messages

Channels

A channel is a destination for a message that is specific to one part in the performance. For example, a channel might receive a note-on message that causes the instrument on that channel to make a sound, or a program-change message that assigns a different instrument to that part. See MIDI Channel Messages.

Under the MIDI 1.0 standard, there are 16 MIDI channels, meaning that no more than 16 instruments can be playing at one time. To support this standard but at the same time make more channels available to applications, DirectMusic creates channel groups. Up to 65,536 channel groups can exist at one time, each containing 16 channels, for a total of over one million channels. A particular port can be assigned any number of channel groups, up to its capability to support them. MIDI hardware ports have only a single channel group.

System-exclusive messages address all 16 channels within a channel group, but not other channel groups.

Every part in a DirectMusic performance has a unique performance channel, sometimes called a PChannel. The performance channel represents a particular MIDI

channel in a particular group on a particular port. When a band is selected by a performance, each instrument in that band is mapped to a performance channel, so the part on that channel will play on that instrument.

When audiopaths are being used, identical performance channels on different audiopaths are mapped to different output channels.

Channel Priority

The number of notes that can be played simultaneously is limited by the number of *voices* available on the port. A voice is a set of resources dedicated to the synthesis of a single note or wave being played on a channel. In the event that more notes are playing than there are available voices, one or more notes must be suppressed by the synthesizer. The choice is determined by the *priority* of the voice currently playing the note, which is based in turn on the priority of the channel. By default, channels are ranked according to their index value, except that channel 10, the MIDI percussion channel, is ranked highest.

[C++]
Applications and synthesizers can set their own channel priorities. For more information, see the Remarks for **IDirectMusicPort8::GetChannelPriority**. See also **DMUS_CHANNEL_PRIORITY_PMSG**.

Message Creation and Delivery

When a segment is played, most of its tracks generate messages containing information about events that are to take place during playback. For more information, see DirectMusic Tracks.

A few tracks send more than one kind of message. For example, a style track sends note messages and time signature messages. In such cases, an application can disable certain kinds of messages in the track. For more information, see Disabling and Enabling Track Parameters.

The performance engine determines when each message is to be processed in clock time. In the case of channel messages, the performance also determines which performance channel gets the message. This information, along with other data—including the message type, its source track, and pointers to the first toolgraph and tool that are to receive it—is stored in the message structure.

Certain messages, such as tempo and time signature changes, are immediately processed and freed by the performance. Other messages, such as notes and patch changes, are placed in a queue and processed in order of time stamp.

Notes

There is no guarantee that messages with the same time stamp will be processed in any particular order.

Time signature messages are purely informational. The time signature is built into the segment and cannot be changed.

[C++]
Messages are first sent to any tools in the segment toolgraph, then to the audiopath toolgraph, and finally to the performance toolgraph.

The application is responsible for creating the segment and performance toolgraphs and defining the tools. For more information on creating toolgraphs, see the following topics:

- **IDirectMusicGraph8**
- **IDirectMusicPerformance8::SetGraph**
- **IDirectMusicSegment8::SetGraph**

Audiopath toolgraphs are optional components of audiopath configurations in DirectMusic Producer files. The application doesn't need to do anything to implement an audiopath toolgraph after creating the audiopath.

The first tool in a toolgraph processes the message and then, if it wants to pass it on, has the toolgraph stamp the message with a pointer to the next tool.

At this point the toolgraph also flags the message with a delivery type that determines when the message is delivered to the next tool. This flag is based on what delivery type the tool is expecting, as follows:

- If the message is flagged as DMUS_PMSGF_TOOL_IMMEDIATE, it is delivered to the next tool immediately.
- If it is flagged as DMUS_PMSGF_TOOL_QUEUE, the message is delivered just before the time at which it is supposed to play, taking latency into account (see Latency and Bumper Time).
- If it is flagged as DMUS_PMSGF_TOOL_ATTIME, it is delivered at exactly the time at which it is to be processed. Notification messages are given this flag, because there is little or no latency involved in processing a notification.

The current tool can change the delivery type after the toolgraph has finished stamping and flagging the message.

Ultimately, unless a message has been discarded, it arrives at the DirectMusic output tool, which converts all the data it receives into standard MIDI messages and delivers these to the synthesizer through the port buffer.

For an illustration, see Overview of Audio Data Flow.

Application-Created Messages

Most messages are generated by tracks, but applications can place messages in the queue directly. You might do this, for example, to change the tempo or to play a sound effect as a note on a DLS instrument.

[C++]
For more information and sample code, see **IDirectMusicPerformance8::SendPMsg**.

When a performance has multiple audiopaths, DirectMusic may not be able to route application-created messages correctly, because a new message contains no information about what audiopath it belongs to. Even a channel-specific message might belong to a performance channel that is mapped to different audiopaths in different segments.

To ensure that a message is sent to a particular audiopath, first obtain the toolgraph for a segment or audiopath by calling **IDirectMusicSegmentState8::GetObjectInPath** or **IDirectMusicAudioPath8::GetObjectInPath**, retrieving the **IDirectMusicGraph8** interface from the DMUS_PATH_AUDIOPATH_GRAPH or DMUS_PATH_SEGMENT_GRAPH stage in the audiopath. Then pass the message to **IDirectMusicGraph8::StampPMsg**.

[Visual Basic]
The following methods send a performance message.

DirectMusicPerformance8.SendCurvePMSG	Curve
DirectMusicPerformance8.SendNotePMSG	Musical note
DirectMusicPerformance8.SendPatchPMSG	Patch change
DirectMusicPerformance8.SendMIDIPMSG	Miscellaneous MIDI events
DirectMusicPerformance8.SendTempoPMSG	Tempo change
DirectMusicPerformance8.SendTimeSigPMSG	Time signature change
DirectMusicPerformance8.SendTransposePMSG	Transposition change

When a performance has multiple audiopaths, DirectMusic may not be able to route application-created messages correctly, because a new message contains no information about what audiopath it belongs to. In DirectX for Visual Basic, there is no way to direct an application-created message to a particular audiopath.

Performance Message Types

[C++]

Messages are stored in various structures derived from **DMUS_PMSG**. Because C does not support inheritance, the members of **DMUS_PMSG** are included in the declaration for each message type as the **DMUS_PMSG_PART** macro. These members contain data common to all messages, including the type of the message, time stamps, the performance channel to which the message is directed, and what toolgraph and tool are next in line to process the message. The other members contain data unique to the message type.

The following standard message structures are defined.

Structure	Content
DMUS_PMSG	Simple message with no additional parameters. See Remarks for the structure.
DMUS_CHANNEL_PRIORITY_PMSG	Channel priority change. See Channels.
DMUS_CURVE_PMSG	Curve.
DMUS_LYRIC_PMSG	Text.
DMUS_MIDI_PMSG	Any MIDI message that does not have a unique message type—for example, a control change.
DMUS_NOTE_PMSG	Music note. (Includes duration, so MIDI note-on and note-off messages are combined in this type.)
DMUS_NOTIFICATION_PMSG	Notification. See Notification and Event Handling.
DMUS_PATCH_PMSG	MIDI patch change.
DMUS_SYSEX_PMSG	MIDI system exclusive message.
DMUS_TEMPO_PMSG	Tempo change.
DMUS_TIMESIG_PMSG	Time signature change.
DMUS_TRANSPOSE_PMSG	Transposition.
DMUS_WAVE_PMSG	Wave playback.

[Visual Basic]
DirectX for Visual Basic does not give applications access to messages after they have been sent, other than notification messages. See Notification and Event Handling.

Curves

A curve is a series of MIDI controller changes bringing about a smooth transition from one value to another—for example, volume fade-out or fade-in.

[C++]
You can execute a curve by sending a single performance message of type **DMUS_CURVE_PMSG**. This structure enables you to set the start and end values, the duration of the curve, and its shape. Optionally, you can also set a reset value, which is the value to which the controller will return in case of an invalidation.

The **wMeasure**, **nOffset**, **bBeat**, **bGrid**, and **mtOriginalStart** members of the message structure are for information only, and do not affect the timing of the message. They are set in messages sent by DirectMusic Producer segments, and can be used by tools. Applications can normally set these members to 0.

The **wMergeIndex** member is used to determine whether changes are cumulative or overriding. Two curve messages with different merge indexes are cumulative; otherwise, each message in turn overrides settings made by a previous message.

The **bCCData** member contains the MIDI controller number for controller changes, and is otherwise ignored. For information on controller numbers, see the MIDI specification.

The following sample code causes the volume to fade from its current value to zero over five seconds. If an invalidation occurs during that period, which might happen if another segment replaces the currently playing segment, full volume is restored.

```
DMUS_CURVE_PMSG *pCurveMsg;
HRESULT hr = g_pPerf->AllocPMsg(sizeof(DMUS_CURVE_PMSG),
    (DMUS_PMSG**) &pCurveMsg);
ZeroMemory(pCurveMsg, sizeof(DMUS_CURVE_PMSG));

pCurveMsg->dwSize = sizeof(DMUS_CURVE_PMSG);
pCurveMsg->rtTime = 0;
pCurveMsg->dwFlags = DMUS_PMSGF_DX8 | DMUS_PMSGF_REFTIME
    | DMUS_PMSGF_LOCKTOREFTIME;
pCurveMsg->dwPChannel = DMUS_PCHANNEL_BROADCAST_PERFORMANCE;
pCurveMsg->dwType = DMUS_PMSGT_CURVE;
pCurveMsg->dwGroupID = 0xFFFFFFF;
pCurveMsg->mtDuration = 5000;
pCurveMsg->nEndValue = 0;
pCurveMsg->bCurveShape = DMUS_CURVES_LINEAR;
pCurveMsg->bCCData = 7;
pCurveMsg->bFlags = DMUS_CURVE_RESET | DMUS_CURVE_START_FROM_CURRENT;
pCurveMsg->bType = DMUS_CURVET_CCCURVE ;

pCurveMsg->mtResetDuration = 0;
pCurveMsg->nResetValue = 127;

g_pPerf->SendPMsg((DMUS_PMSG*) pCurveMsg);
```

Note

A simpler way to implement volume fading is by using **IDirectMusicAudioPath8::SetVolume**. This method always uses the linear shape for the curve.

[Visual Basic]
You can execute a curve by sending a single performance message, using **DirectMusicPerformance8.SendCurvePMSG**. The **DMUS_CURVE_PMSG** type passed to this method enables you to set the start and end values, the duration of the curve, and its shape. Optionally, you can also set a reset value, which is the value to which the controller will return in case of an invalidation.

The **measure**, **offset**, **beat**, **grid**, and **mtOriginalStart** members of the message structure are used in messages sent by DirectMusic Producer segments. In messages sent by applications, these values should be 0.

The **MergeIndex** member is used to determine whether changes are cumulative or overriding. Two curve messages with different merge indexes are cumulative; otherwise, each message in turn overrides settings made by a previous message.

The **ccData** member contains the MIDI controller number for controller changes, and is otherwise ignored. For information on controller numbers, see the MIDI specification.

The following sample code fades out the current volume over a duration of 5 seconds:

```
Dim curveMsg As DMUS_CURVE_PMSG

curveMsg.curveShape = DMUS_CURVES_LINEAR
curveMsg.mtDuration = 5000
curveMsg.Type = DMUS_CURVET_CCCURVE
curveMsg.ccData = 7  ' MIDI channel volume controller
curveMsg.flags = DMUS_CURVE_START_FROM_CURRENT

dmPerf.SendCurvePMSG 0, _
  DMUS_PMSGF_REFTIME Or DMUS_PMSGF_LOCKTOREFTIME, _
  DMUS_PCHANNEL_BROADCAST_PERFORMANCE, curveMsg
```

Note

A simpler way to implement volume fading is by using **DirectMusicAudioPath8.SetVolume**. This method always uses the linear shape for the curve.

The following code changes the pitch bend from the current value to the maximum value over a duration of 3 seconds. If an invalidation occurs during that period, which might happen if another segment replaces the currently playing segment, the pitch is reset to the value in *g_Pitch*.

```
curveMsg.curveShape = DMUS_CURVES_LINEAR
```

```
curveMsg.startValue = 0  ' Ignored, start from current.
curveMsg.endValue = 16383
curveMsg.mtDuration = 3000
curveMsg.Type = DMUS_CURVET_PBCURVE
curveMsg.ccData = 0
curveMsg.flags = DMUS_CURVE_RESET Or DMUS_CURVE_START_FROM_CURRENT
curveMsg.resetValue = g_Pitch
curveMsg.mtResetDuration = 0

dmPerf.SendCurvePMSG 0, _
  DMUS_PMSGF_REFTIME Or DMUS_PMSGF_LOCKTOREFTIME, _
  DMUS_PCHANNEL_BROADCAST_PERFORMANCE, curveMsg
```

MIDI Messages

Most applications don't deal directly with MIDI messages. However, an application can send a MIDI command as a performance message—for example, to make a control change.

MIDI messages consist of a status byte and usually one or two data bytes. System exclusive MIDI messages are of variable length.

The status byte indicates the type of message and, in some cases, the channel that is to receive the message. When several events of the same kind are in sequence in the file, the status byte can be omitted. Data bytes are recognizable because the high bit is always clear, whereas in status bytes it is always set.

The timing of MIDI events is controlled by a number before each message, indicating how many ticks separate this event from the last. The actual duration of a tick depends on the time format in the file header.

Note

There is no guarantee that MIDI messages will be processed in the same order in which they occur in the source data. DirectMusic messages are delivered in order of time stamp, and two MIDI messages with identical times might not be delivered in the expected order. Care must be taken, in authoring MIDI content, to leave an interval between events if they must take place sequentially. For example, don't place a program change at the same time as a note that depends on the program change.

MIDI messages are divided into two main categories:

- MIDI Channel Messages
- MIDI System Messages

MIDI Channel Messages

A channel message is addressed to a particular MIDI channel, which corresponds to a single part in the music.

A channel message can be either a *mode message* or a *voice message*.

A mode message determines how a channel will deal with subsequent voice messages. For example, a mode message might instruct the channel to remain silent, ignoring all note-on messages until further notice.

Most channel messages are voice messages. They instruct the channel to begin or stop playing a note or to modify the note in some way, or they change the timbre by assigning a different MIDI patch number to the channel.

The following table describes types of voice messages.

Voice message	Purpose
Note-on	Play a note.
Note-off	Stop playing the note.
Control change	Modify the tone with data from a pedal, lever, or other device; also used for miscellaneous controls such as volume and bank select.
Program (patch) change	Select an instrument for the channel by assigning a patch number.
Aftertouch	Modify an individual note, or all notes on the channel, according to the aftertouch of a key.
Pitch bend change	Modify the pitch of all notes played on the channel.

Keep in mind that these descriptions apply to standard MIDI messages, not MIDI data that has been converted to performance message format. For example, two MIDI messages to start and stop a note are combined by DirectMusic into a single performance message giving the duration of the note. DirectMusic messages also contain much additional information about timing and routing.

MIDI Notes

The data bytes of a note-on message represent the pitch and velocity. In most cases, a pitch value of 0 represents C below subcontra C (called C0 in MIDI notation), 12 represents subcontra C (or C1), 60 is middle C (or C5), and so on. For drum kits, the data byte instead represents a particular drum sound. For example, as long as the General MIDI (GM) percussion key map is being adhered to, a value of 60 represents a high bongo sound. Channel 10 is reserved for drum kits, so the synthesizer knows that note-on messages on that channel are to be treated differently than on other channels.

For information on how DirectMusic converts to and from MIDI notes, see Music Values and MIDI Notes.

Program Changes

Program changes and patch numbers are a key concept in MIDI playback and in DirectMusic. A program change assigns a particular instrument (also called a program or timbre) to a channel so that the notes sent to that channel are played with the appropriate sound. Instruments are identified by patch numbers. If the GM instrument set is loaded, a program change specifying patch number 1 always causes the channel to play its notes as an acoustic grand piano. The actual sound produced at the speakers depends on how the instrument is synthesized.

Bank Selection

Because a single data byte is used to select the patch number in a program change, and only 7 bits in each data byte of a MIDI message are significant, a program change can select from a maximum of 128 instruments. To provide a greater choice, the MIDI specification allows for the use of up to 16,384 instrument banks, each containing up to 128 instruments.

To select an instrument from a different bank, the MIDI sequencer must first send a control change message called *bank select*. The 2 data bytes of this message are referred to as the most significant byte (MSB) and least significant byte (LSB), and they are combined to identify a bank. Once the bank has been selected, each subsequent program change selects an instrument from that bank.

DirectMusic Patch Numbers

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not enable applications to change patch numbers except by using the **DirectMusicPerformance8.SendPatchPMSG** method.

[C++]
In DirectMusic, the instrument patch number is not the 7-bit MIDI patch number but a 32-bit value that packs the MIDI patch number together with the MSB and LSB of the bank select and a one-bit flag for a drum kit. This extended patch number is returned by the **IDirectMusicCollection8::EnumInstrument**, **IDirectMusicCollection8::GetInstrument**, and **IDirectMusicInstrument8::GetPatch** methods. It can be changed for an instrument by using the **IDirectMusicInstrument8::SetPatch** method.

The organization of DirectMusic patch values is shown in the following table.

Bits	Purpose
0-7	MIDI patch number (bit 7 is always 0)
8-15	LSB bank select (bit 15 is always 0)
16-23	MSB bank select (bit 23 is always 0)
24-30	Unused

31	Flag for drum kit

MIDI System Messages

System messages are not exclusive to any channel. There are three kinds, as shown in the following table.

Message type	Purpose
System common	Miscellaneous commands and data.
System exclusive	Equipment-specific commands and data.
System real-time	Synchronization of clock-based MIDI equipment.

Unlike other MIDI messages, system exclusive messages can contain any number of data bytes. After transmitting the data, the sequencer sends a system common message called an EOX, which signals the end of the system exclusive message.

[C++]
In DirectMusic, the **DMUS_SYSEX_PMSG** structure contains the length of the data and a pointer to an array of data bytes.

[Visual Basic]
DirectX for Visual Basic does not give applications direct access to system exclusive messages.

Using DirectMusic Ports

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not support direct access to ports.

[C++]
This section covers access to DirectMusic ports by applications that do not use audiopaths. If your application initializes the performance by using **IDirectMusicPerformance8::InitAudio**, as is recommended, the audiopath manages ports and the mapping of performance channels to ports.

A port is a device that sends or receives data. It can correspond to a hardware device, a software synthesizer, or a software filter.

Each port in a DirectMusic application is represented by an **IDirectMusicPort8** interface. Methods of this interface are used to retrieve information about the device, manage the memory on the device, download and unload DLS instruments, read incoming data, and cue playback buffers.

Every performance must have at least one port. If you want to use a port other than the default port, or to set special parameters for the default port, first set up a **DMUS_PORTPARAMS8** structure. You don't have to fill in all members, but you must let DirectMusic know which members have valid information by putting the appropriate flags in the **dwValidParams** member. Then pass the structure to the **IDirectMusic8::CreatePort** method.

The following C++ code demonstrates how an object might be created for the default port, setting five channel groups on the port. Assume that *pDirectMusic* is a valid **IDirectMusic8** interface pointer.

```
IDirectMusicPort8*  pPort;
DMUS_PORTPARAMS    dmos;

ZeroMemory( &dmos, sizeof(DMUS_PORTPARAMS) );
dmos.dwSize = sizeof(DMUS_PORTPARAMS);
dmos.dwValidParams = DMUS_PORTPARAMS_CHANNELGROUPS;
dmos.dwChannelGroups = 5;
HRESULT hr = pDirectMusic->CreatePort( GUID_NULL, &dmos,
      &pPort, NULL )
```

Once you have a port, you must activate it by calling **IDirectMusic8::Activate** or **IDirectMusicPort8::Activate** and attach it to the performance by using the **IDirectMusicPerformance8::AddPort** method.

When you add a port to a performance, assign a block of PChannels to it by calling the **IDirectMusicPerformance8::AssignPChannelBlock** method. The only time this isn't necessary is when you add the default port by passing NULL to **IDirectMusicPerformance8::AddPort**. In that case, PChannels 0 through 15 are assigned to the MIDI channels in the first group on the port.

You can map PChannels differently, add more PChannels, or assign PChannels to a different port by using the **IDirectMusicPerformance8::AssignPChannelBlock** and **IDirectMusicPerformance8::AssignPChannel** methods.

More information about ports is contained in the following topics:

- Default Port
- Property Sets for DirectMusic Ports

Default Port

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not support direct access to ports.

[C++]
Under Windows® 95 and versions of Windows 98 prior to Windows 98 Second Edition, and always when hardware that supports DLS is not available, the Microsoft software synthesizer is the default port. Under later versions of Windows 98 and under Windows 2000, a hardware synthesizer could be the default port.

If you want your application to use the default port, you don't have to call the **IDirectMusic8::CreatePort** method before adding the port to the performance. Instead, you can pass NULL to **IDirectMusicPerformance8::AddPort**.

You can obtain the default port by a call to **IDirectMusic8::GetDefaultPort**, and then check its capabilities by using the **IDirectMusicPort8::GetCaps** method. If the port does not meet the needs of your application, use the **IDirectMusic8::EnumPort** method to find the Microsoft software synthesizer or another port.

Custom Loading

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic supports loading only of standard DirectMusic objects.

[C++]
Specialized applications might create their own object types that encapsulate data from a file or resource. It can be convenient to have the DirectMusic loader handle the loading of such objects. This is especially true if the custom object is referenced by other objects or contains references to other objects.

To implement a loading mechanism that takes advantage of the DirectMusic loader, take the following steps:

- Register the object class so that it can be found by **IDirectMusicLoader8::GetObject**.
- Implement the **IDirectMusicObject8** interface on the object so that the loader can get the information that it needs to find and cache it.

- Implement the **IPersistStream** interface on the object, with full functionality in the **IPersistStream::Load** method. This is where you parse the data that you obtain through calls on the **IStream** interface passed by the DirectMusic loader.
- In the implementation of **Load**, ensure that references to other objects are dealt with by querying the **IStream** for the **IDirectMusicGetLoader8** interface, then calling **IDirectMusicGetLoader8::GetLoader** to obtain a pointer to the DirectMusic loader that created the stream. Once you have this pointer, you can call **IDirectMusicLoader8::GetObject** to load the new object.

It might happen that an application needs to manage file input itself—for example, because all objects are stored in a special compressed resource file. The application can create its own loader by creating an object that supports the **IDirectMusicLoader8** interface, with the **IDirectMusicLoader8::GetObject** method implemented. All other methods are optional. This implementation of the loader must also create its own stream object that has both the **IStream** and the **IDirectMusicGetLoader8** interfaces.

Using Instrument Collections

In most applications, DLS instrument data is associated with bands and is downloaded to the synthesizer when the band is downloaded. For more information, see Using Bands.

[C++]
For specialized DirectMusic applications that do their own DLS management, two steps must be taken: loading the instrument collection and downloading instrument data to a port.

These steps are covered in the following sections:

- Loading and Downloading Collections
- Working with Instruments

Applications for editing instruments and collections must work with DLS data at an even lower level. This topic is covered in the following section:

- Low-Level DLS

[Visual Basic]
Information about working directly with instrument collections is presented in the following topic:

- Loading and Downloading Collections

Loading and Downloading Collections

[C++]
Collections are loaded like other objects. For more information, see Loading Audio Data.

To load the standard GM/GS set, pass GUID_DefaultGMCollection to the loader in the **guidObject** member of the **DMUS_OBJECTDESC** structure. If you intend to use the loader to access this object more than once, make sure that caching is enabled (as it is by default) so that you don't create another copy of the GM collection each time you request it.

Note

The GM/GS Sound Set cannot be altered. For more information, see the Copyright Warning.

The following code example illustrates how to load a collection identified by its GUID:

```
HRESULT myGetGMCollection(
    IDirectMusicLoader8 *pILoader,
    IDirectMusicCollection8 **ppICollection)
{
  HRESULT hr;
  DMUS_OBJECTDESC desc;

  desc.dwSize = sizeof(DMUS_OBJECTDESC);
  desc.guidClass = CLSID_DirectMusicCollection;
  desc.guidObject = GUID_DefaultGMCollection;
  desc.dwValidData = (DMUS_OBJ_CLASS | DMUS_OBJ_OBJECT);
  hr = pILoader->GetObject(&desc, IID_IDirectMusicCollection8,
      (void **) ppICollection);
  return hr;
}
```

Once you have obtained a pointer to the **IDirectMusicCollection8** interface, you have access to all the instruments in the collection. At this point, though, none of them have been downloaded to a port.

To download an entire collection at once, you must associate the collection with a segment and then call the **IDirectMusicSegment8::Download** method. For an example, see Playing a MIDI File with Custom Instruments.

These steps are necessary only when you want to use a collection other than the default one. Normally, when you call **IDirectMusicSegment8::Download**, the instruments downloaded to the port are from the default collection authored into the

segment, or from the General MIDI set if the segment does not reference a collection. When you download a band, all DLS data needed by the instruments in that band is downloaded. See Using Bands.

[Visual Basic]
To load an instrument collection, call the **DirectMusicLoader8.LoadCollection** or the **DirectMusicLoader8.LoadCollectionFromResource** method. Each of these methods returns a **DirectMusicCollection8** object. This object has no methods, and its only role is to be passed as a parameter to **DirectMusicSegment8.ConnectToCollection**. You call that method to associate the collection with a segment, and then download the instruments by calling **DirectMusicSegment8.Download**. For an example, see Playing a MIDI File with Custom Instruments.

These steps are necessary only when you want to use a collection other than the default one. Normally, when you call **DirectMusicSegment8.Download**, the instruments downloaded to the port are from the default collection authored into the segment, or from the General MIDI set if the segment does not reference a collection. When you download a band, all DLS data needed by the instruments in that band is downloaded. See Using Bands.

Working with Instruments

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not enable applications to work with individual instruments from a collection.

[C++]
When a collection object is created and loaded from a collection file or resource, it is not bound to any specific port. You can download different instruments to different ports or download a single instrument to multiple ports.

You can retrieve the patch number and name of all the available instruments by using the **IDirectMusicCollection8::EnumInstrument** method.

The following code example enumerates all instruments in a collection and displays their names and patch numbers:

```
void myListInstruments(
    IDirectMusicCollection8 *pCollection)

{
  HRESULT hr = S_OK;
  DWORD dwPatch;
```

```
    WCHAR wszName[MAX_PATH];
    DWORD dwIndex;
    for (dwIndex = 0; hr == S_OK; dwIndex++)
    {
       hr = pCollection->EnumInstrument(
             dwIndex, &dwPatch, wszName, MAX_PATH);
       if (hr == S_OK)
       {
         printf("Patch %lx is %S\n",dwPatch,wszName);
       }
    }
}
```

Obtain a pointer to a specific instrument by passing its patch number to the **IDirectMusicCollection8::GetInstrument** method.

After obtaining an instrument, you can change its patch number by using the **IDirectMusicInstrument8::SetPatch** method.

To download a single instrument to a port, pass an **IDirectMusicInstrument8** interface pointer to the **IDirectMusicPort8::DownloadInstrument** or **IDirectMusicPerformance8::DownloadInstrument** method. This call makes the DLS data available on the port; it does not associate the instrument with any particular performance or audiopath.

To save memory, only waves and articulation required for given ranges of notes are downloaded. For example, for a bassoon you might specify that only data for the note range from low C through middle B is to be downloaded. Data for regions falling entirely outside that range is not downloaded.

The following code example, given a collection, a patch number, a port, and a range of notes, retrieves the instrument from the collection and downloads it. It sets up an array of one **DMUS_NOTERANGE** structure and passes this to the **IDirectMusicPort8::DownloadInstrument** method. Typically, only a single range of notes is specified, but it is possible to specify multiple ranges. If you pass NULL instead of an array, the data for all regions is downloaded.

```
HRESULT myDownload(
    IDirectMusicCollection8 *pCollection,    // DLS collection
    IDirectMusicPort8 *pPort,                // Destination port
    IDirectMusicDownloadedInstrument8 **ppDLInstrument,
    DWORD dwPatch,                           // Requested instrument
    DWORD dwLowNote,                         // Low note of range
    DWORD dwHighNote)                        // High note of range

{
    HRESULT hr;
    IDirectMusicInstrument8* pInstrument;
    hr = pCollection->GetInstrument(dwPatch, &pInstrument);
    if (SUCCEEDED(hr))
```

```
    {
        DMUS_NOTERANGE NoteRange[1];        // Optional note range
        NoteRange[0].dwLowNote = dwLowNote;
        NoteRange[0].dwHighNote = dwHighNote;
        hr = pPort->DownloadInstrument(pInstrument,
                ppDLInstrument,
                NoteRange,   // Array of ranges
                1);         // Number of elements in the array
        pInstrument->Release();
    }
    return hr;
}
```

The **DownloadInstrument** method returns a pointer to the **IDirectMusicDownloadedInstrument8** interface. This pointer has just one purpose: to identify the instrument in a subsequent call to the **IDirectMusicPort8::UnloadInstrument** method, which unloads the instance of the instrument on a particular port.

The following function downloads an instrument and then unloads it, which is not useful except to illustrate how the **IDirectMusicDownloadedInstrument8** pointer can be used:

```
HRESULT myFickleDownload(
        IDirectMusicInstrument8* pInstrument,
        IDirectMusicPort8 *pPort,
        DWORD dwPatch)

{
    HRESULT hr;
    IDirectMusicDownloadedInstrument * pDLInstrument;
    hr = pPort->DownloadInstrument(
            pInstrument, &pDLInstrument,
            NULL, 0);
    if (SUCCEEDED(hr))
    {
        pPort->UnloadInstrument(pDLInstrument);
        pDLInstrument->Release();
    }
    return hr;
}
```

Low-Level DLS

[Visual Basic]

This topic pertains only to applications written in C++. DirectX for Visual Basic does not support low-level manipulation of DLS data.

[C++]
If you are writing an application that edits DLS collections, you need to be able to download instrument data to the synthesizer without encapsulating it in a DirectMusic instrument object.

Working with DLS data requires knowledge of the DLS specification and file structure. For detailed information on these topics, contact the MIDI Manufacturers Association.

To download raw instrument data, you must first get a pointer to the **IDirectMusicPortDownload8** interface, as shown in the following code example, where it is assumed that *pIPort* is a valid pointer to an **IDirectMusicPort8** interface:

```
IDirectMusicPortDownload **ppIDownloadPort;

HRESULT hr = pIPort->QueryInterface(IID_IDirectMusicPortDownload8,
    (void **) ppIDownloadPort);
```

If the **HRESULT** is not S_OK, the port does not support DLS downloading.

Next, identify the buffers that must be prepared and downloaded. To send an instrument to the synthesizer, create one instrument buffer that represents the entire instrument definition with all the regions and articulations, and a series of wave buffers, one for each wave that the instrument references for its regions.

Each buffer must be tagged with a unique identifier. Identifiers are used to resolve linkages between buffers, in particular the links between regions and waves. Tally the number of buffers that you need to download, and call **IDirectMusicPortDownload8::GetDLId** to allocate a range of identifiers. For example, if you are downloading an instrument with three waves, you must download four buffers in all, so request a set of four identifiers.

For each buffer, calculate the size needed; then call **IDirectMusicPortDownload8::AllocateBuffer** to allocate it. This method returns an **IDirectMusicDownload8** interface representing the buffer. Call **IDirectMusicDownload8::GetBuffer** to access the memory.

Note

There are two methods called **GetBuffer**:
IDirectMusicPortDownload8::GetBuffer returns an **IDirectMusicDownload** interface pointer for a buffer object whose download identifier is known.
IDirectMusicDownload::GetBuffer returns a pointer to the memory in the buffer.

Now write the data into the buffers. Each buffer starts with a **DMUS_DOWNLOADINFO** structure, which defines the size and functionality of the download. This structure must be prepared as follows:

- Set the **dwDLType** member to either DMUS_DOWNLOADINFO_INSTRUMENT2 for an instrument or DMUS_DOWNLOADINFO_WAVE for a wave.
- Set the **dwDLId** member to one of the unique identifiers that you obtained by using **IDirectMusicPortDownload::GetDLId**.
- Set the **dwNumOffsetTableEntries** member to the number of entries in the **DMUS_OFFSETTABLE** structure.
- Set the **cbSize** member to the size of the download chunk, including **DMUS_DOWNLOADINFO** and **DMUS_OFFSETTABLE**.

The **DMUS_DOWNLOADINFO** structure is always followed by a **DMUS_OFFSETTABLE** structure. This offset table is used to manage all links within the data. Whenever a structure in the data refers to another structure, it addresses it with an integer index instead of a pointer. For every structure within the data that can be referenced, there is a unique index. The **DMUS_OFFSETTABLE** translates this integer index into a byte offset into the data.

The instrument or wave data follows the **DMUS_OFFSETTABLE**. If the download is an instrument, the data starts with the **DMUS_INSTRUMENT** structure. Otherwise, it starts with the **DMUS_WAVE** structure.

The instrument data that follows the **DMUS_INSTRUMENT** structure is organized in the following structures:

- **DMUS_ARTICPARAMS**
- **DMUS_ARTICULATION**
- **DMUS_ARTICULATION2**
- **DMUS_COPYRIGHT**
- **DMUS_EXTENSIONCHUNK**
- **DMUS_INSTRUMENT**
- **DMUS_NOTERANGE**
- **DMUS_REGION**

The wave data pointed to by the **DMUS_WAVE** structure is organized in a **DMUS_WAVEDATA** structure.

When the buffers are all ready, download them by using **IDirectMusicPortDownload8::Download**. Download the wave buffers first so that they are in place and can be referenced when the instrument is downloaded.

Once the buffers have been downloaded, the synthesizer is ready to play the instrument. The memory in the buffer is no longer accessible.

Later, when done playing the instrument, unload the buffers and release them. First unload the instrument buffer, then all the wave buffers. To unload, pass the **IDirectMusicDownload8** pointers to **IDirectMusicPortDownload8::Unload**. Then release each buffer with a call to **IDirectMusicDownload8::Release**.

To update an instrument that has already been downloaded, you cannot write over the previously downloaded buffer. Instead, replace the instrument, but not the waves. To do this, call **IDirectMusicPortDownload8::AllocateBuffer** to allocate a new **IDirectMusicDownload8** interface with a buffer of the correct size. Be sure to generate a new identifier for the buffer with a call to **IDirectMusicPortDownload8::GetDLId**. Write the new articulation information into the buffer; then download it. Then unload the previously downloaded buffer with a call to **IDirectMusicPortDownload8::Unload**.

To update a wave buffer, take one extra step. Create both a new wave buffer and an updated instrument buffer that references it. Download the new wave, then the new instrument. Then unload the old instrument and the old wave.

DirectMusic Tools

[Visual Basic]
This topic pertains only to applications written in C++. DirectX for Visual Basic does not support the use of application-defined or third-party tools.

[C++]
A tool is an object that intercepts messages and handles them in some way. The tool might alter the message and then pass it on to the next tool, or free the message, or send a new message based on information in the old one.

DirectMusic has an output tool that is normally the last to receive messages. It is this tool that converts performance messages to standard MIDI messages and streams them to the synthesizer. Other tools are implemented by the application or obtained from libraries.

To implement a tool, you must first create an object that supports the **IDirectMusicTool8** interface. The object's implementation of the **IDirectMusicTool8** methods determines what messages are processed by the tool and what work is performed on these messages.

All tools other than the output tool are collected in toolgraphs. Even if your application is using only one other tool, you must create a toolgraph to contain it. Then add this toolgraph to a segment or the performance. Toolgraphs provide a convenient mechanism for directing messages from one tool to another.

When the performance engine is playing a segment, it enables each tool in the segment toolgraph, and then each tool in the performance toolgraph, to process each

message. After a tool processes a message, it should obtain the **IDirectMusicGraph8** pointer from the **pGraph** member of the **DMUS_PMSG** structure and then call the **IDirectMusicGraph8::StampPMsg** method to stamp the message with a pointer to the next tool, if any, that is to receive it.

Tools process messages in a high-priority thread. Do not call time-consuming functions, such as those involving graphics or file input/output, from within a tool's **IDirectMusicTool8::ProcessPMsg** method. If a tool needs to trigger an action, it should do so by signaling a different thread, perhaps the application's main thread.

When implementing the methods of **IDirectMusicTool8**, take care not to create circular references to parent objects. Circular references come about when one object creates another and the child keeps an additional reference to the parent. For example, suppose a tool creates a new reference to the toolgraph passed into its **IDirectMusicTool8::Init** method. If the tool fails to release this reference, there is a problem when the segment attempts to release the toolgraph. Because the tool still has a reference to the toolgraph, the toolgraph is not fully released; and because the toolgraph has a reference to the tool, the tool cannot be released either.

For an example of how to implement a tool, see the MusicTool sample application.

Property Sets

[Visual Basic]
This topic applies only to applications written in C++. DirectX for Visual Basic does not provide access to property sets.

[C++]
Through property sets, DirectX Audio is able to support extended services offered by ports and sound cards.

Hardware vendors define new capabilities as properties and publish the specification for these properties. A **GUID** identifies a property set, and a **ULONG** identifies a particular property within the set. For example, a hardware vendor might design a card capable of reverberation effects and define a property set DSPROPSETID_ReverbProps containing properties such as DSPROPERTY_REVERBPROPS_HALL and DSPROPERTY_REVERBPROPS_STADIUM. Typically, the property identifiers are defined using a C language enumeration starting at ordinal zero.

Individual properties may also have associated parameters. The meaning of the parameters is defined along with the properties.

Property Sets for DirectMusic Ports

[Visual Basic]
This topic applies only to applications written in C++. DirectX for Visual Basic does not provide access to property sets.

[C++]
Use the **IKsControl::KsProperty** method to find out whether a property is available and then to set and retrieve values for that property. You obtain the **IKsControl** interface for a port by calling the **IDirectMusicPort8::QueryInterface** method, passing IID_IKsControl as the interface identifier.

A property set is represented by a GUID, and each item within the set is represented by a zero-based index. The meaning of the indexed items for a GUID never changes. For a list of the property sets supported by DirectMusic, see **KSPROPERTY**.

All property sets predefined by DirectMusic have only one item, usually at index 0. However, the full definition of kernel-streaming (KS) properties is supported, and vendors are free to create property sets with any number of items and instances, and data of any size.

Routing of the property item request to the port varies depending on the port implementation. No properties are supported by ports that represent DirectMusic emulation over the Win32® handle-based multimedia calls (the **midiOut** and **midiIn** functions).

The following code example uses the **IKsControl::KsProperty** method to determine if the port supports General MIDI in hardware:

```
BOOL IsGMSupported(IDirectMusicPort8 *pPort)
{
    HRESULT     hr;
    IKsControl  *pControl;
    KSPROPERTY  ksp;
    DWORD       dwFlags;
    ULONG       cb;
    BOOL        fIsSupported;

    // Query for an IKsControl interface.
    hr = pPort->QueryInterface(IID_IKsControl, (void**)&pControl);
    if (FAILED(hr))
    {
        // Port does not support properties; assume no GM support.
        return FALSE;
    }
    // Ask about GM.
    ksp.Set   = GUID_DMUS_PROP_GM_Hardware;
```

```
    ksp.Id    = 0;
    ksp.Flags = KSPROPERTY_TYPE_BASICSUPPORT;
    hr = pControl->KsProperty(&ksp, sizeof(ksp),
          &dwFlags, sizeof(dwFlags), &cb);
    fIsSupported = FALSE;
    if (SUCCEEDED(hr) || (cb >= sizeof(dwFlags))
    {
       // Set is supported.
       fIsSupported = (BOOL)(dwFlags & KSPROPERTY_TYPE_GET);
    }
    pControl->Release();
    return fIsSupported;
}
```

Property Sets for DirectSound Buffers

[Visual Basic]
This topic applies only to applications written in C++. DirectX for Visual Basic does not provide access to property sets.

[C++]
To make use of extended properties on sound cards, you must first determine whether the driver supports the **IKsPropertySet** interface, and obtain a pointer to the interface if it is supported. You can do this by calling the **QueryInterface** method of an existing interface on a DirectSound 3-D buffer object, as follows:

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

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

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

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

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

Optimizing DirectSound Performance

This section offers some miscellaneous tips for improving the performance of applications that play their audio data directly into DirectSound buffers. The following topics are covered:

- Matching Buffer Formats
- Using Hardware Mixing
- Dynamic Voice Management
- Hardware Acceleration on ISA and PCI Cards
- Playing the Primary Buffer Continuously
- Minimizing Control Changes
- CPU Considerations for 3-D Buffers

Matching Buffer Formats

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

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

Using Hardware Mixing

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

- At run time, use the **IDirectSound8::GetCaps** method to determine what formats are supported by the sound-accelerator hardware and use only those formats if possible.

- Create sound buffers first for the sounds you use the most. There is a limit to the number of buffers that can be mixed by hardware.
- To force a buffer to be created in hardware, specify the DSBCAPS_LOCHARDWARE flag in the **dwFlags** member of the **DSBUFFERDESC** structure. If you do this and resources for hardware mixing are not available, the **IDirectSound8::CreateSoundBuffer** method will fail.
- Use voice management to enable DirectSound to assign buffers to hardware resources when they are available, and to force termination of less important buffers in order to free hardware resources. See Dynamic Voice Management.

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

- At run time, use the **DirectSound8.GetCaps** method to determine what formats are supported by the sound-accelerator hardware and use only those formats if possible.
- Create sound buffers first for the sounds you use the most. There is a limit to the number of buffers that can be mixed by hardware.
- To force a buffer to be created in hardware, specify the DSBCAPS_LOCHARDWARE flag in the **lFlags** member of the **DSBUFFERDESC** type. If you do this and resources for hardware mixing are not available, the **DirectSound8.CreateSoundBuffer** method will fail.
- Use voice management to enable DirectSound to assign buffers to hardware resources when they are available, and to force termination of less important buffers in order to free hardware resources. See Dynamic Voice Management.

Dynamic Voice Management

Many sound cards maintain their own secondary buffers and handle 3-D effects and mixing for them. These hardware buffers, as they are called, usually reside in system memory rather than on the card itself, but because they are mixed by the device, they make far smaller demands on the system processor than do software buffers. It is therefore most efficient to have as many buffers as possible allocated to hardware, especially 3-D buffers.

By default, DirectSound allocates buffers to hardware whenever it can. However, it can create only as many hardware buffers as the device can play at one time—that is, the number of hardware buffers is limited by the hardware voices available. DirectSound allocates a hardware voice when a buffer is created and releases it only when the buffer is destroyed; the voice is not free even when the buffer is not playing.

If an application creates many buffers, chances are that some of them will end up being in software—that is, they will be managed and mixed by the CPU rather than the sound card.

Dynamic voice management helps you overcome the limited availability of hardware resources by deferring voice allocation until buffers are played, and by enabling DirectSound to stop lower-priority sounds prematurely so that their resources can be allocated to new sounds.

[C++]
To defer the allocation of resources for hardware mixing and 3-D effects to the moment when the buffer is played, specify the DSBCAPS_LOCDEFER flag in the **DSBUFFERDESC** structure passed to **IDirectSound8::CreateSoundBuffer**. When you call **IDirectSoundBuffer8::Play** or **IDirectSoundBuffer8::AcquireResources** on a buffer created with the DSBCAPS_LOCDEFER flag, DirectSound places the buffer in hardware if possible, and in software otherwise.

[Visual Basic]
To defer the allocation of resources for hardware mixing and 3-D effects to the moment when the buffer is played, specify the DSBCAPS_LOCDEFER flag in the **DSBUFFERDESC** type passed to **DirectSound8.CreateSoundBuffer**, **DirectSound8.CreateSoundBufferFromFile**, or **DirectSound8.CreateSoundBufferFromResource**. When you call **DirectSoundSecondaryBuffer8.Play** or **DirectSoundSecondaryBuffer8.AcquireResources** on a buffer created with the DSBCAPS_LOCDEFER flag, DirectSound places the buffer in hardware if possible, and in software otherwise.

When calling **Play**, you can attempt to free a hardware voice for the buffer by passing one of the flags in the following table, or one of the first two flags in combination with the third. DirectSound then searches for a playing buffer that is suitable for termination as specified by the flags.

Flag	Criterion
DSBPLAY_TERMINATEBY_TIME	Select the buffer that has been playing longer than any other candidate buffers.
DSBPLAY_TERMINATEBY_DISTANCE	Select the 3-D candidate buffer farthest from the listener.
DSBPLAY_TERMINATEBY_PRIORITY	Select the buffer that has the lowest priority of candidate buffers, as set in the call to **Play**. If this is combined with one of the other two flags, the other flag is used only to resolve ties.

If it finds a suitable buffer, DirectSound stops it and allocates the voice it was using to the newly played buffer. If no buffer is suitable for termination, the newly played buffer is played in software, unless the DSBPLAY_LOCHARDWARE flag was specified, in which case the call to **Play** fails.

[C++]
For more information on how deferred buffers are dealt with when played, see **IDirectSoundBuffer8::Play**.

[Visual Basic]
For more information on how deferred buffers are dealt with when played, see **DirectSoundSecondaryBuffer8.Play**.

Note

Do not use the DSBCAPS_STATIC flag in combination with DSBCAPS_LOCDEFER when creating buffers. DSBCAPS_STATIC buffers are placed in on-card memory when this is available, as is the case for some PCI cards. If you create such a buffer with the DSBCAPS_LOCDEFER flag, the buffer memory is not copied to the sound card until the buffer is played for the first time, which can result in unacceptable delays. For more information, see Voice Management on ISA and PCI Cards.

Hardware Acceleration on ISA and PCI Cards

In the past few years, sound cards have moved from the ISA bus to the PCI bus. This design change has fundamentally altered the way DirectSound-accelerated cards handle audio data. In addition, DirectSound has introduced voice management, which enables applications to make more efficient use of limited hardware acceleration resources.

This section discusses some considerations in allocating buffers and using voice management on the two kinds of sound cards. The following topics are discussed:

- DirectSound Buffers on ISA Cards
- DirectSound Buffers on PCI Cards
- Voice Management on ISA and PCI Cards

DirectSound Buffers on ISA Cards

When DirectSound was first introduced, all audio devices operated on the ISA bus. The ISA bus was sufficient to stream 16-bit, 22 kHz stereo data from the CPU to the sound card without unduly affecting overall system performance, but attempting to send more data often placed an undue burden on the bus. This limitation had a strong influence on audio hardware design.

To mix DirectSound buffers, a chip on the sound card has two choices: it can reach across the ISA bus into system memory, or it can have dedicated memory on the card itself. Owing to the ISA bus's limited bandwidth, reading audio data from system memory is not practical. Therefore, ISA-based sound cards that do DirectSound acceleration almost always have dedicated on-card RAM.

When such a sound card is used to accelerate DirectSound, the sound data must first be copied from system memory, across the ISA bus, and into the dedicated memory on the card. This copying can take some time. However, once the data is on the card, there is almost no cost to the system for starting, stopping, and mixing sounds; all that is handled by the audio processor chip accessing its own dedicated memory.

ISA-based DirectSound accelerators support static buffers on the card, but not streaming buffers, because it is impractical to stream data across the slow ISA bus. DirectSound buffers created with the DSBCAPS_STATIC flag are located in card memory if it is available. Others are created in system memory.

DirectSound Buffers on PCI Cards

Most audio cards sold today, if not all, are designed to connect to the PCI bus. With the movement to this bus comes a large increase in the available bandwidth between the system and the audio card. It is now practical to have the audio chip reach directly into system memory to retrieve audio data for hardware mixing. As a result, it is no longer worthwhile to have on-board memory for buffers.

On PCI cards, static buffers and streaming buffers are effectively the same. The DSBCAPS_STATIC flag does not affect the location of the buffer, because all buffers are located in system memory.

[C++]

Note

Most sound cards report in **DSCAPS** that they have static buffers, even though they have no on-card RAM. The streaming and static buffers reported by these cards are often the same buffers.

[Visual Basic]

Note

Most sound cards report in **DSCAPS** that they have static buffers, even though they have no on-card RAM. The streaming and static buffers reported by these cards are often the same buffers.

Voice Management on ISA and PCI Cards

By using the DSBCAPS_LOCDEFER flag when creating a buffer, you can defer the allocation of buffers to hardware-mixed or software-mixed memory until the moment when they are played. For information, see Dynamic Voice Management.

The effect of deferred allocation on application performance depends on the type of DirectSound accelerator in the user's system.

Consider the case of a PCI accelerator. When a deferred buffer is played, DirectSound first determines if there is hardware available to play the sound. Assuming there is, DirectSound then ensures that the hardware has access to the sound data, which remains where it is. Finally, DirectSound tells the hardware to start playing. This process is both efficient and fast.

Note

The DirectSound voice manager allocates hardware mixing resources, not memory. On a PCI card, the buffer occupies the same memory before and after it is allocated, regardless of whether it is allocated to the hardware mixer or the software mixer.

If the user has an ISA-based accelerator, the situation is quite different. In this case, the act of getting audio data to a hardware-mixed buffer is very slow, because the data needs to cross the ISA bus and be loaded into memory on the sound card before it can be mixed. For deferred buffers, this can introduce an unacceptable delay between when the play call is made and when the sound starts. For this reason, using voice management is not recommended on ISA-based DirectSound accelerator cards.

It is easy to prevent your application from using voice management on ISA devices. A buffer won't be assigned to a on-card memory unless the DSBCAPS_STATIC flag is set. Therefore, as long as you do not combine DSBCAPS_STATIC and DSBCAPS_LOCDEFER, there is no danger that buffers will be assigned to on-card memory at play time.

Using DSBCAPS_STATIC by itself ensures that the buffer will take advantage of any existing memory on an ISA card, but has no effect on a PCI card, where in the absence of other flags, DirectSound will attempt to place the buffer under hardware control but still in system memory. Using DSBCAPS_LOCDEFER by itself has no effect on an ISA card, since the buffer is always placed in system memory and is not managed by the hardware.

The following table summarizes the effects of the DSBCAPS_STATIC and DSBCAPS_LOCDEFER flags on buffers created for ISA and PCI devices.

Card	Flags	Memory	HW acceleration
ISA	DSBCAPS_STATIC	Hardware if available.	Yes, if in hardware memory.
PCI	DSBCAPS_STATIC	System.	Yes, if hardware voice available; flag has no effect.
ISA	DSBCAPS_LOCDEFER	Hardware if available, if DSBCAPS_STATIC	Yes, if in hardware memory; but latency on first play.

		set.	
PCI	DSBCAPS_LOCDEFER	System.	Yes, if hardware voice available.

To take advantage of acceleration on ISA cards without running into the dangers presented by deferred allocation, first check the capabilities of the device. If the card reports more than zero static buffers and no streaming buffers, it is probably an older ISA card with on-card memory. You can then use the DSBCAPS_STATIC flag, but do not use DSBCAPS_LOCDEFER. If the card reports more than zero streaming buffers, it is probably a newer PCI-based accelerator card. In this case, you'll want to use DSBCAPS_LOCDEFER, and not DSBCAPS_STATIC, to make maximum use of the hardware voices.

Playing the Primary Buffer Continuously

When there are no sounds playing, DirectSound stops the mixer engine and halts DMA (direct memory access) activity. If your application has frequent short intervals of silence, the overhead of starting and stopping the mixer each time a sound is played may be worse than the DMA overhead if you kept the mixer active. Also, some sound hardware or drivers may produce unwanted audible artifacts from frequent starting and stopping of playback.

[Visual Basic]
If your application is playing audio almost continuously with only short breaks of silence, you can force the mixer engine to remain active by calling the **DirectSoundPrimaryBuffer8.Play** method.

To resume the default behavior of stopping the mixer engine when there are no sounds playing, call the **DirectSoundPrimaryBuffer8.Stop** method.

To obtain the primary buffer object, use the **DirectSound8.CreatePrimarySoundBuffer** method.

[C++]
If your application is playing audio almost continuously with only short breaks of silence, you can force the mixer engine to remain active by calling the **IDirectSoundBuffer8::Play** method for the primary buffer.

To resume the default behavior of stopping the mixer engine when there are no sounds playing, call the **IDirectSoundBuffer8::Stop** method for the primary buffer.

Minimizing Control Changes

Performance is affected when you change the pan, volume, or frequency on a secondary buffer. To prevent interruptions in sound output, the DirectSound mixer

must mix ahead from 20 to 100 or more milliseconds. Whenever you make a control change, the mixer has to flush its mix-ahead buffer and remix with the changed sound.

[C++]
It's a good idea to minimize the number of control changes you send. Try reducing the frequency of calls to routines that use the **IDirectSoundBuffer8**::**SetVolume**, **IDirectSoundBuffer8::SetPan**, and **IDirectSoundBuffer8::SetFrequency** methods. For example, if you have a routine that moves a sound from the left to the right speaker in synchronization with animation frames, try calling the **SetPan** method only every second or third frame.

[Visual Basic]
It's a good idea to minimize the number of control changes you send. Try reducing the frequency of calls to routines that use the **DirectSoundSecondaryBuffer8.SetVolume**, **DirectSoundSecondaryBuffer8.SetPan**, and **DirectSoundSecondaryBuffer8.SetFrequency** methods. For example, if you have a routine that moves a sound from the left to the right speaker in synchronization with animation frames, try calling the **SetPan** method only every second or third frame.

Note

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

CPU Considerations for 3-D Buffers

Software-emulated 3-D buffers are computationally expensive. You should take this into consideration when deciding when and how to use 3-D buffers in your applications.

[C++]
Use as few 3-D sounds as you can, and don't use 3-D on sounds that won't really benefit from the effect. Design your application so that it's easy to enable and disable 3-D effects on each sound. You can call the **IDirectSound3DBuffer8::SetMode** method with the DS3DMODE_DISABLE flag to disable 3-D processing on any 3-D sound buffer.

[Visual Basic]

Use as few 3-D sounds as you can, and don't use 3-D on sounds that won't really benefit from the effect. Design your application so that it's easy to enable and disable 3-D effects on each sound. You can call the **DirectSound3DBuffer8.SetMode** method with the DS3DMODE_DISABLE flag to disable 3-D processing on any 3-D sound buffer.

You can also ensure that 3-D processing is suspended on buffers that are too far away from the listener to be heard. See Minimum and Maximum Distances.

DirectSound Driver Models

Under the VxD driver model, all DirectSound mixing is done in Dsound.vxd, a virtual device driver. Dsound.vxd also provides close access to the actual DMA buffer that the sound card uses to receive data from the host CPU. This is the same as the DirectSound primary buffer. A DirectSound application can set specific properties of the primary buffer, such as sampling rate and bit depth, thus changing the properties of the hardware itself.

Under the Windows Driver Model (WDM), DirectSound does not have direct access to the sound hardware, except in the case of hardware-accelerated buffers. Instead, DirectSound talks to the kernel mixer, or Kmixer. Kmixer's job is to convert the format of multiple audio streams to a common format, mix them together and send the result to the hardware. In a sense, it does what Dsound.vxd does. One major difference is that Dsound.vxd only mixes DirectSound buffer data, but Kmixer mixes all Windows audio data, including data from applications that use the Win32 **waveOut** functions. The rule that DirectSound and the waveform-audio output device can't both be open at the same time is not true on systems with WDM drivers.

Of particular importance is Kmixer's relationship with the audio hardware. Kmixer is the only software on the system that can specify the format of the hardware's DMA buffer. It selects the format on the basis of sounds it is asked to mix. It sets the output format to the highest-quality format of sounds that it is asked to mix, or to the closest format to this that the hardware supports.

This has one very important implication: DirectSound cannot set the format of the hardware's DMA buffer. For your application, this means that the hardware format is based on the data you actually play. If you play a 44 kHz file, Kmixer will mix all data up to 44 kHz and ensure that the hardware is running at 44 kHz.

As an application developer, you don't choose the driver model used. That is completely determined by the type of sound card, the version of Windows, and the particular driver the user has installed. For that reason, it is very important that you cover all possibilities when you test your application. DirectSound might be using Dsound.vxd or it might be using Kmixer, and you should ensure your application's behavior and performance are acceptable on both.

Writing to the Primary Buffer

[Visual Basic]
Applications can write directly to the primary buffer by using the **DirectSoundPrimaryBuffer8.WriteBuffer** method, provided that the cooperative level is set to DSSCL_WRITEPRIMARY. However, it is strongly recommended that DirectX for Visual Basic applications not do so.

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

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

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

Note

Writing directly to the primary buffer offers no advantages under the WDM driver model. Under WDM, the primary buffer is in effect a secondary buffer that is mixed by the kernel mixer. For more information on WDM, see DirectSound Driver Models.

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

You cannot obtain write access to a primary buffer unless it exists in hardware. To determine whether this is the case, call the **IDirectSoundBuffer8::GetCaps** method and check for the DSBCAPS_LOCHARDWARE flag in the **dwFlags** member of the **DSBCAPS** structure that is returned. If you attempt to lock a primary buffer that is emulated in software, the call will fail.

You create an accessible primary buffer by specifying the DSBCAPS_PRIMARYBUFFER flag in the **DSBUFFERDESC** structure passed to the **IDirectSound8::CreateSoundBuffer** method. If you want to write to the buffer, the cooperative level must be DSSCL_WRITEPRIMARY.

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

The following example shows how to obtain write access to the primary buffer. Note that the primary buffer supports only the **IDirectSoundBuffer** interface, not **IDirectSoundBuffer8**.

```
BOOL AppCreateWritePrimaryBuffer(
    LPDIRECTSOUND8 lpDirectSound,
    LPDIRECTSOUNDBUFFER *lplpDsb,
    LPDWORD lpdwBufferSize,
    HWND hwnd)
{
    DSBUFFERDESC dsbdesc;
    DSBCAPS dsbcaps;
    HRESULT hr;
    WAVEFORMATEX wf;

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

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

    // Obtain write-primary cooperative level.
    hr = lpDirectSound->SetCooperativeLevel(hwnd, DSSCL_WRITEPRIMARY);
    if SUCCEEDED(hr)
    {
        // Try to create buffer.
        hr = lpDirectSound->CreateSoundBuffer(&dsbdesc,
                lplpDsb, NULL);
        if SUCCEEDED(hr)
        {
            // Set primary buffer to desired format.
            hr = (*lplpDsb)->SetFormat(&wf);
            if SUCCEEDED(hr)
            {
```

```
            // If you want to know the buffer size, call GetCaps.
               dsbcaps.dwSize = sizeof(DSBCAPS);
            (*lplpDsb)->GetCaps(&dsbcaps);
            *lpdwBufferSize = dsbcaps.dwBufferBytes;
            return TRUE;
         }
      }
   }
   // Failure.
   *lplpDsb = NULL;
   *lpdwBufferSize = 0;
   return FALSE;
}
```

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

```
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->Lock(dwOldPos, dwDataBytes,
                     &lpvPtr1, &dwBytes1,
                     &lpvPtr2, &dwBytes2, 0);

   // If DSERR_BUFFERLOST is returned, restore and retry lock.

   if (DSERR_BUFFERLOST == hr)
   {
      lpDsbPrimary->Restore();
      hr = lpDsbPrimary->Lock(dwOldPos, dwDataBytes,
                        &lpvPtr1, &dwBytes1,
                        &lpvPtr2, &dwBytes2, 0);
```

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

Programming Tips and Tools

This section contains information to help you develop DirectX Audio applications efficiently. The following topics are covered:

- Debugging DirectX Audio Projects
- Adjusting DirectSound Acceleration for Testing

Debugging DirectX Audio Projects

[Visual Basic]
This topic pertains only to application development in C++.

[C++]
The DirectMusic and DirectSound dynamic-link libraries (DLLs) installed with the debug version of the DirectX software development kit generate information in the debug output window as the application is running. By default, all available information is shown. These DLLs are available if you installed the debug version of

the DirectX SDK. For DirectMusic, they can be dynamically selected through the DirectX property sheet in Control Panel by choosing **Use Debug Version of DirectMusic**.

For DirectMusic, you can control the volume of information that goes to your debug output window by changing values in Win.ini. The output for each DirectMusic DLL can be set separately, as in the following example:

```
[Debug]
DMBAND=1
DMCOMPOS=1
DMIME=1
DMLOADER=0
DMUSIC=1
DMSTYLE=3
DMSYNTH=5
```

Each value can be in the range from -1 through 5, where -1 produces no debugging information and 5 the most detailed information. If there is no entry in Win.ini, the debug output is at level 0. You can focus on problems in a particular DLL by setting lower values for the other components.

You can also set the debug level within the range from 0 to 5 by using the **Debug Output Level** sliders on the DirectMusic and DirectSound pages of the DirectX property sheet in Control Panel. However, this method sets the same value for all DLLs.

See also Debugging DirectX Applications.

Adjusting DirectSound Acceleration for Testing

By using the Sounds and Multimedia Properties page in Control Panel, you can adjust the performance of DirectSound on the system. Click the **Advanced** button in the **Sound Playback** group on the **Audio** tab. Then choose the **Performance** tab. A slider lets you adjust hardware acceleration to one of the following settings:

Full
: Enables complete DirectSound acceleration, including the enabling of **IKsPropertySet** extensions. This is the default setting on Windows 98.

Standard
: Enables acceleration of DirectSound secondary buffers, but disables any hardware-specific **IKsPropertySet** extensions. This is the default setting on Windows 2000.

Basic

Disables hardware acceleration of DirectSound secondary buffers. This option is useful if you want to emulate a nonaccelerated sound card for testing purposes.

Emulation

Forces DirectSound into emulation mode, where it acts as if there is no DirectSound-compatible driver on the system. All mixing is done by DirectSound in user mode, and the resulting data is sent to the Win32 waveform-audio functions. This typically results in a large increase in latency.

DirectX Audio C++ Tutorials

This section contains the following tutorials showing how to implement Microsoft® DirectMusic® and Microsoft® DirectSound® in a C++ application:

- Tutorial 1: Playing Audio Files
- Tutorial 2: Using Audiopath Objects

Other, more specialized uses of Microsoft® DirectX® Audio are demonstrated in the sample applications provided with the SDK. For a description of these samples, see DirectX Audio C++ Samples.

Tutorial 1: Playing Audio Files

This tutorial is a step-by-step guide to the most basic tasks in DirectX Audio: initializing a DirectMusic performance and playing an audio file. The tutorial is presented in the following steps:

- Step 1: Initialize
- Step 2: Load a File
- Step 3: Play the File
- Step 4: Close Down

The complete sample code for the tutorial is available in the following folder:

```
\mssdk\samples\Multimedia\DirectMusic\Tutorials\Tut1
```

Note

If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "mssdk" in the path.

Step 1: Initialize

The following includes are needed for any application that uses the DirectMusic API. Including Dmusici.h also cause the other necessary header files for DirectMusic and DirectSound to be included.

```
#define INITGUID
#include <dmusici.h>
```

The tutorial uses three interface pointers, which are declared as follows:

```
IDirectMusicLoader8*      g_pLoader      = NULL;
IDirectMusicPerformance8* g_pPerformance  = NULL;
IDirectMusicSegment8*     g_pSegment     = NULL;
```

All the code in this simple application is included in the **WinMain** function. The application has no main window, so it can proceed straight to the creation of COM and two objects: the loader and the performance:.

```
INT APIENTRY WinMain( HINSTANCE hInst, HINSTANCE hPrevInst,
    LPSTR pCmdLine, INT nCmdShow )
{
    CoInitialize(NULL);

    CoCreateInstance(CLSID_DirectMusicLoader, NULL,
                CLSCTX_INPROC, IID_IDirectMusicLoader8,
                (void**)&g_pLoader);

    CoCreateInstance(CLSID_DirectMusicPerformance, NULL,
                CLSCTX_INPROC, IID_IDirectMusicPerformance8,
                (void**)&g_pPerformance );
```

The next step is to initialize the performance and the synthesizer. The **IDirectMusicPerformance8::InitAudio** method performs the following tasks:

- Creates a DirectMusic and a DirectSound object. In most cases you don't need an interface to those objects, and you can pass NULL in the first two parameters.
- Associates an application window with the DirectSound object. Normally the handle of the main application window is passed as the third parameter, but the tutorial application doesn't have a window, so it passes NULL instead.
- Sets up a default audiopath of a standard type. The tutorial requests a path of type DMUS_APATH_SHARED_STEREOPLUSREVERB, which is suitable for music.
- Allocates a number of performance channels to the audiopath. Wave files require only a single performance channel, and MIDI files require up to 16. Segments created in DirectMusic Producer might need more. No harm is done by asking for extra channels.

- Specifies capabilities and resources of the synthesizer. This can be done in one of two ways: by setting flags or by supplying a **DMUS_AUDIOPARAMS** structure with more detailed information. Most applications set the DMUS_AUDIOF_ALL flag and let DirectMusic create the synthesizer with default parameters.

In the tutorial, the call to **InitAudio** is very simple:

```
g_pPerformance->InitAudio(
    NULL,               // IDirectMusic interface not needed.
    NULL,               // IDirectSound interface not needed.
    NULL,               // Window handle.
    DMUS_APATH_SHARED_STEREOPLUSREVERB,  // Default audiopath type.
    64,                 // Number of performance channels.
    DMUS_AUDIOF_ALL,    // Features on synthesizer.
    NULL                // Audio parameters; use defaults.
);
```

Next: Step 2: Load a File

Step 2: Load a File

The DirectMusic performance and synthesizer are now ready to process sound data. To get the data, the loader needs to know where to find it. Although a full path can be provided each time a file is loaded, it is more convenient to establish a default directory. Do this by using the **IDirectMusicLoader8::SetSearchDirectory** method.

In the sample code, the path to the default Windows media directory is given. You can change the value of *wstrSearchPath* to get files from a different folder.

The following code is from the **WinMain** function in the tutorial sample:

```
// Find the Windows media directory.

CHAR strPath[MAX_PATH];
GetWindowsDirectory( strPath, MAX_PATH );
strcat( strPath, "\\media" );

// Convert to Unicode.

WCHAR wstrSearchPath[MAX_PATH];
MultiByteToWideChar( CP_ACP, 0, strPath, -1,
                     wstrSearchPath, MAX_PATH );

// Set the search directory.

g_pLoader->SetSearchDirectory(
    GUID_DirectMusicAllTypes,   // Types of files sought.
    wstrSearchPath,             // Where to look.
```

```
    FALSE                   // Don't clear object data.
);
```

In the call to **SetSearchDirectory**, the *fClear* parameter is set to FALSE because there is no danger of accidentally reloading objects from the wrong directory. This is likely to happen only if the application is loading identically named objects from different folders.

Now that the loader knows where to look for the file, it can load it as a segment:

```
WCHAR wstrFileName[MAX_PATH] = L"The Microsoft Sound.wav";

if (FAILED(g_pLoader->LoadObjectFromFile(
    CLSID_DirectMusicSegment,   // Class identifier.
    IID_IDirectMusicSegment8,   // ID of desired interface.
    wstrFileName,               // Filename.
    (LPVOID*) &g_pSegment       // Pointer that receives interface.
)))
{
    MessageBox( NULL, "Media not found, sample will now quit.",
                "DMusic Tutorial", MB_OK );
    return 0;
}
```

Next: Step 3: Play the File

Step 3: Play the File

The wave file loaded in the previous step is now available to the performance through its **IDirectMusicSegment8** interface.

Before a segment can be played, its band must be downloaded to the synthesizer. As long as you don't unload the band, this step has to be taken only once for each segment that uses a unique band.

The following code from the **WinMain** function in the sample downloads the band to the performance. Alternatively, it could be downloaded to an audiopath. As long as only a single synthesizer is in use, it doesn't matter which destination object you choose:

```
g_pSegment->Download( g_pPerformance );
```

To play the file, pass the segment interface to **IDirectMusicPerformance8::PlaySegmentEx**. This method offers many options for playback, but to play a segment immediately on the default audiopath, all the parameters except the first can be NULL or 0:

```
g_pPerformance->PlaySegmentEx(
    g_pSegment,  // Segment to play.
```

```
    NULL,       // Used for songs; not implemented.
    NULL,       // For transitions.
    0,          // Flags.
    0,          // Start time; 0 is immediate.
    NULL,       // Pointer that receives segment state.
    NULL,       // Object to stop.
    NULL        // Audiopath, if not default.
);
MessageBox( NULL, "Click OK to Exit.", "Play Audio", MB_OK );
```

Next: Step 4: Close Down

Step 4: Close Down

To exit an audio application cleanly, you must perform four main steps:

- Stop any playing segments by calling **IDirectMusicPerformance8::Stop**.
- Close down the performance. The **IDirectMusicPerformance8::CloseDown** method performs miscellaneous cleanup tasks and releases internal references to objects.
- Release all interfaces.
- Close COM.

The following sample code from the **WinMain** function in the tutorial sample is called when the dialog box is closed.

```
    g_pPerformance->Stop(
        NULL,   // Stop all segments.
        NULL,   // Stop all segment states.
        0,      // Do it immediately.
        0       // Flags.
    );

    g_pPerformance->CloseDown();

    g_pLoader->Release();
    g_pPerformance->Release();
    g_pSegment->Release();

    CoUninitialize();

    return 0;  // Return value for WinMain.
}           // End of WinMain.
```

Tutorial 2: Using Audiopath Objects

This tutorial is a guide to setting up a DirectMusic performance and retrieving an object—in this case, a 3-D buffer—from an audiopath so that sound parameters can be changed. The tutorial is presented in the following steps:

- Step 1: Create the Audiopath
- Step 2: Retrieve the Buffer
- Step 3: Change Buffer Parameters

The complete sample code for the tutorial is available in the following folder:

```
\mssdk\samples\Multimedia\DirectMusic\Tutorials\Tut2
```

Note
If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "mssdk" in the path.

It is assumed that you have already learned the basic steps of creating the performance and loader objects, and loading and playing a file. These steps are covered in Tutorial 1: Playing Audio Files.

Step 1: Create the Audiopath

The simplest way to create an audiopath is by passing a flag to **IDirectMusicPerformance8::InitAudio**. The tutorial sample passes the DMUS_APATH_DYNAMIC_STEREO flag, causing **InitAudio** to set up a default audiopath that supports stereo sounds:

```
g_pPerformance->InitAudio(
    NULL,                   // IDirectMusic interface not needed.
    NULL,                   // IDirectSound interface not needed.
    NULL,                   // Window handle.
    DMUS_APATH_DYNAMIC_STEREO, // Default audiopath type.
    64,                     // Number of performance channels.
    DMUS_AUDIOF_ALL,        // Features on synthesizer.
    NULL                    // Audio parameters; use defaults.
);
```

The default audiopath is suitable for sounds that do not have to be located in space, such as background music or narration. However, if an application implements 3-D sound effects, it will play each sound source on its own audiopath, so that 3-D parameters can be set individually.

The sample creates one such audiopath as follows:

```
IDirectMusicAudioPath8* p3DAudioPath = NULL;
g_pPerformance->CreateStandardAudioPath(
    DMUS_APATH_DYNAMIC_3D,      // Path type.
```

```
    64,                     // Number of performance channels.
    TRUE,                   // Activate now.
    &p3DAudioPath           // Pointer that receives audiopath.
);
```

A segment can now be played on this audiopath as follows:

```
g_pPerformance->PlaySegmentEx(
    g_pSegment,  // Segment to play.
    NULL,       // Used for songs; not implemented.
    NULL,       // For transitions.
    0,          // Flags.
    0,          // Start time; 0 is immediate.
    NULL,       // Pointer that receives segment state.
    NULL,       // Object to stop.
    p3DAudioPath // Audiopath.
);
```

Next: Step 2: Retrieve the Buffer

Step 2: Retrieve the Buffer

By using the **IDirectMusicAudioPath8::GetObjectInPath** method, you can retrieve interfaces to objects that form part of the path. In the case of the DMUS_APATH_DYNAMIC_3D standard audiopath type, such objects could include the secondary buffer itself, the primary buffer, the DirectSound listener, or any DMOs set on buffers after the audiopath was created. The tutorial sample obtains the **IDirectSound3DBuffer8** interface to the buffer:

```
IDirectSound3DBuffer8* pDSB = NULL;

p3DAudioPath->GetObjectInPath(
    DMUS_PCHANNEL_ALL,      // Performance channel.
    DMUS_PATH_BUFFER,       // Stage in the path.
    0,                  // Index of buffer in chain.
    GUID_NULL,              // Class of object.
    0,                  // Index of object in buffer; ignored.
    IID_IDirectSound3DBuffer, // GUID of desired interface.
    (LPVOID*) &pDSB         // Pointer that receives interface.
);
```

The parameters to **IDirectMusicAudioPath8::GetObjectInPath** can be a little tricky to set up properly. For information on which parameters are relevant for objects at different stages in the path, see Retrieving Objects from an Audiopath.

In this case, you are retrieving a secondary buffer that is used by all performance channels on this audiopath. Set the *dwPChannel* parameter to DMUS_PCHANNEL_ALL.

Because the buffer you want is the first and in this case the only buffer in the chain, you pass 0 as *dwBuffer*. The DMUS_PATH_BUFFER stage contains only buffer objects, and not the DMOs attached to those buffers; therefore *dwIndex* is ignored.

Next: Step 3: Change Buffer Parameters

Step 3: Change Buffer Parameters

Now that you have the **IDirectSound3DBuffer8** interface, you can use it to move the sound in space. The tutorial sample application does so when the user closes a series of message boxes. For example, the following code immediately moves the sound to the left:

```
pDSB->SetPosition( -0.1f, 0.0f, 0.0f, DS3D_IMMEDIATE );
```

The first three parameters specify the new position of the sound source in relation to the default listener. The default listener is at coordinates (0.0, 0.0, 0.0), facing toward the positive z-axis, with the top of the head toward the positive y-axis. Distance units are meters by default. Because the x-axis is positive from left to right, the new position of the sound is 10 centimeters directly to the left of the listener. For more information, see Coordinates of 3-D Space and Listener Orientation.

The last parameter of the **IDirectSound3DBuffer8::SetPosition** method specifies whether the change is to be made immediately or deferred until all changes are committed. For more information, see Deferred Settings.

DirectX Audio C++ Samples

The sample applications described in this section demonstrate the use and capabilities of the Microsoft® DirectMusic® and Microsoft® DirectSound® application programming interfaces (APIs) in Microsoft® DirectX® for C++.

The samples are described under the following headings:

- DirectMusic C++ Samples. These samples use the DirectMusic API to load and play sounds. Some of them also use elements of the DirectSound API.
- DirectSound C++ Samples. These samples use only the DirectSound API. However, many of the techniques shown can also be used with buffers obtained from a DirectMusic audiopath.

DirectMusic C++ Samples

The following executable applications using the DirectMusic API, or DirectSound and DirectMusic together, are found on the **Start** menu under **Programs/Microsoft DirectX8 SDK/DirectMusic Samples**:

- 3DAudio
- AudioFX
- AudioPath
- AudioScripts
- MusicTool
- PlayAudio
- PlayMotif

The source code is in the following folder:

```
\mssdk\samples\Multimedia\DirectMusic
```

In addition to these samples, the source files for tutorial applications are contained in the following folder:

```
\mssdk\samples\Multimedia\DirectMusic\Tutorials
```

For more information, see DirectX Audio C++ Tutorials.

The samples other than the tutorials use common source files that implement functions and classes for basic DirectMusic and DirectSound functionality and for general tasks such as finding media files. These files are found in the following folder:

```
\mssdk\samples\Multimedia\Common\src
```

Note

If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "mssdk" in all paths.

3DAudio

Description

The 3DAudio sample application shows how to create a 3-D audiopath in a DirectMusic performance, how to obtain an interface to a 3-D buffer and listener in that path, and how to modify the parameters of the buffer and listener.

Path

Source: (SDK Root)\Sample\Multimedia\DirectMusic\3DAudio

Executable: (SDK Root)\Sample\Multimedia\DirectMusic\Bin

User's Guide

Click **Open File** and load a wave, MIDI, or DirectMusic segment file. Play the segment. The position of the sound source is shown as a red dot on the graph, where the x-axis is from left to right and the z-axis is from bottom to top. Change the range of movement on the two axes by using the sliders.

The listener is located at the center of the graph, and has its default orientation, looking along the positive z-axis; that is, toward the top of the screen. The sound source moves to the listener's left and right and to the listener's front and rear, but does not move above and below the listener.

The sliders in the center of the window control the properties of the listener; that is, the global sound properties. If you click **Defer Settings**, changes are not applied until you click **Apply Settings**.

Programming Notes

See Readme.txt in the source code folder.

See Also

3-D Sound, Retrieving Objects from an Audiopath

AudioFX

Description

The AudioFX sample shows how to use DMOs on DirectMusic audiopaths to add effects to sounds, and how to set effect parameters.

Path

Source: (SDK Root)\Sample\Multimedia\DirectMusic\AudioFX

Executable: (SDK Root)\Sample\Multimedia\DirectMusic\Bin

User's Guide

A default sound file is loaded when the application is run. If you wish, load a different one by clicking **Open File**.

At first, no effects are enabled. Click **Play** to hear the sound without effects.

Click **Stop** to stop the buffer. Apply one or more effects by selecting checkboxes in the **Enable** column. Play the sound again.

To adjust parameters for an effect, select an option button in the **Adjust** column and change the values in the frame on the right side of the window. This can be done regardless of whether the sound is playing and regardless of whether the effect has been applied yet.

Programming Notes

The application implements a CSoundFXManager class to manage effects. In the CSoundFXManager::Initialize method, it retrieves an **IDirectSoundBuffer8** interface from the audiopath. This interface is used to set effects on the buffer in the CSoundFXManager::ActivateFX method. Effect parameters are set in the OnEffectChanged function in response to messages from the interface.

See Also

Using Effects

AudioPath

Description

The AudioPath sample demonstrates how different sounds can be played on an audiopath, and how the parameters of all sounds are affected by changes made on the audiopath.

Path

Source: (SDK Root)\Sample\Multimedia\DirectMusic\AudioPath

Executable: (SDK Root)\Sample\Multimedia\DirectMusic\Bin

User's Guide

Click **Lullaby**, **Snore**, and **Mumble** to play different sounds. Adjust the 3-D position of the sounds by using the sliders. Click **Rude Awakening** to play a different sound and stop all other sounds.

Programming Notes

This sample is very similar in form to the PlayAudio sample. The AudioPath differs by showing some of the various uses of an audiopath.

On WM_INITDIALOG, the OnInitDialog function does the following:

1. Calls IDirectMusicPerformance8::CreateStandardAudioPath, passing in DMUS_APATH_DYNAMIC_3D to create a 3-D audiopath represented by the **IDirectMusicAudioPath8** interface *g_p3DAudiopath*.

2. Uses the CMusicManager framework class to create CMusicSegment objects from a list of files.
3. Gets the **IDirectSound3DListener8** interface from the audiopath.
4. Calls **IDirectSound3DListener8::SetRolloffFactor** to change the rate at which the amplitude of sounds diminishes over distance.

When the 3-D position slider is changed, the SetPosition function does the following:

1. Calls **IDirectMusicAudioPath8::GetObjectInPath** to retrieve the **IDirectSound3DBuffer8** interface.
2. Calls **IDirectSound3DBuffer8::SetPosition** to set the position of the buffer.
3. Releases the buffer.

When a segment is played, the PlaySegment function does one of the following:

- If the **Lullaby** button was clicked, the segment is played on the audiopath as the primary segment.
- If **Snore** or **Mumble** was clicked, a secondary segment is played.
- If **Rude Awakening** was clicked, all sounds on the audiopath are stopped because the audiopath is passed to **IDirectMusicPerformance8::PlaySegmentEx** as the *pFrom* parameter. The alarm sound is then played as a new primary segment.

See Also

Using Audiopaths

AudioScripts

Description

The AudioScripts sample demonstrates how an application and a DirectMusic script work together. The script reads and writes to variables in the application, and the application calls routines in the script that play segments.

The sample also demonstrates how waves can be played as variations in a segment.

Path

Source: (SDK Root)\Sample\Multimedia\DirectMusic\AudioScripts

Executable: (SDK Root)\Sample\Multimedia\DirectMusic\Bin

User's Guide

Select ScriptDemoBasic.spt from the **Script File** list box. Play a segment by clicking **Routine 1**. Click **Routine 2** to play an ending and stop playback. Play the segment

again and click **Routine 3** several times. Note how **Variable 1** reflects the number of times the button has been clicked, and how the music changes in response to each click.

Select ScriptDemoBaseball.spt from the **Script File** list box. Click **Routine 1** to play various calls from a vendor. Click **Routine 2** to play various musical motifs. Change the score by entering different values in the **Variable 1** and **Variable 2** text boxes. Click **Routine 3** to hear the score.

Programming Notes

See Readme.txt in the source code folder.

See Also

Using Audio Scripts

MusicTool

Description

The MusicTool sample demonstrates how to implement a DirectMusic tool that intercepts messages.

Path

Source: (SDK Root)\Sample\Multimedia\DirectMusic\MusicTool

Executable: (SDK Root)\Sample\Multimedia\DirectMusic\Bin

User's Guide

Play the default segment, or choose another wave, MIDI, or DirectMusic segment file by clicking **Open File**. Apply the echo effect by selecting **Echo Tool** from the drop-down list.

Programming Notes

See Readme.txt in the source code folder.

See Also

DirectMusic Tools

PlayAudio

Description

The PlayAudio sample shows how to load a segment and play it on an audiopath, how to use DirectMusic notifications, and how to change global performance parameters.

Path

Source: (SDK Root)\Sample\Multimedia\DirectMusic\PlayAudio

Executable: (SDK Root)\Sample\Multimedia\DirectMusic\Bin

User's Guide

Play the default segment, or load another wave, MIDI, or DirectMusic segment file by clicking **Open File**. Adjust the tempo and volume by using the sliders.

Programming Notes

On WM_INITDIALOG, the OnInitDialog function does the following:

1. Creates a Win32® event, *g_hDMusicMessageEvent*. This will be used by DirectMusic to signal the application whenever a DirectMusic notification comes in.
2. Creates an object of class CMusicManager called *g_pMusicManager*.
3. Initializes the CMusicManager object. This does the following:
   - Creates **IDirectMusicLoader8** by using **CoCreateInstance**.
   - Creates **IDirectMusicPerformance8** by using **CoCreateInstance**.
   - Calls **IDirectMusicPerformance8::InitAudio** to initialize the performance and create a standard audio path.
4. Calls **IDirectMusicPerformance8::AddNotificationType**,requesting notifications of type GUID_NOTIFICATION_SEGMENT. DirectMusic will notify the application of all segment events so it can ascertain when the segment has ended.
5. Calls **IDirectMusicPerformance8::SetNotificationHandle**, passing in the Win32 event *g_hDMusicMessageEvent*. This tells DirectMusic to signal this event when a notification is available.

The WinMain function performs the following tasks:

1. Creates the window by using **CreateDialog**.
2. In the messasge loop, calls **MsgWaitForMultipleObjects**, passing in *g_hDMusicMessageEvent*. This will tell us when *g_hDMusicMessageEvent* is signaled. DirectMusic signals this event whenever a DirectMusic notification has come in.
3. If WAIT_OBJECT_0 is returned, calls ProcessDirectMusicMessages.

4. If WAIT_OBJECT_0 + 1 is returned, Windows messages are available. The function does standard message processing by using **PeekMessage**.

When **Open File** is clicked, the OnOpenSoundFile function performs the following tasks:

1. Gets the file name.
2. Releases any previously created segment.
3. Calls CMusicManager::CollectGarbage in Dmutil.cpp. This calls **IDirectMusicLoader8::CollectGarbage**, which ensures that unused objects are released. See Garbage Collection.
4. Calls CMusicManager::SetSearchDirectory. This calls **IDirectMusicLoader8::SetSearchDirectory**, passing in GUID_DirectMusicAllTypes and a directory. This tells DirectMusic where to look for files referenced by segments.
5. Calls CMusicManager::CreateSegmentFromFile to create a CMusicSegment called *g_pMusicSegment*. This entails the following steps:
   - Call **IDirectMusicLoader8::LoadObjectFromFile** to load the **IDirectMusicSegment8** into *pSegment*.
   - Create a CMusicSegment, passing in *pSegment*.
   - If the file is a pure MIDI file, call **IDirectMusicSegment8::SetParam**, passing in GUID_StandardMIDIFile. This ensures that the file is played correctly.
   - Call **IDirectMusicSegment8::Download**, which downloads the segment's bands to the synthesizer. Some applications might want to wait before downloading, because the more instruments are downloaded, the more memory is required.

When **Play** is clicked, the OnPlayAudio function does the following:

1. If the sound is to be looped, calls CMusicSegment::SetRepeats, passing in DMUS_SEG_REPEAT_INFINITE. Otherwise repeats are set to zero.
2. Call CMusicSegment::Play, which calls **IDirectMusicPerformance8::PlaySegmentEx**.

When a notification is signaled, the ProcessDirectMusicMessages function looks for a message indicating that a segment has stopped. It performs the following tasks:

1. Calls **IDirectMusicPerformance8::GetNotificationPMsg** in a loop to process each available message. The loop tests for S_OK, because S_FALSE is returned when no more messages are available.
2. If the **dwNotificationOption** of the **DMUS_NOTIFICATION_PMSG** structure is DMUS_NOTIFICATION_SEGEND, calls **QueryInterface** on the **punkUser** member to obtain the **IDirectMusicSegmentState8** interface of the segment instance that ended. The segment itself is obtained by using **IDirectMusicSegmentState8::GetSegment**. This method returns **IDirectMusicSegment**, and **QueryInterface** must be used to obtain

IDirectMusicSegment8. The application then compares this point to the global primary segment pointer, to ensure that it was indeed the primary segment that stopped. Segments authored in DirectMusic Producer can trigger other segments, so we can't be sure that only the primary segment was playing.

3. Cleans up all the interfaces.

PlayMotif

Description

The PlayMotif sample demonstrates how a motif played as a secondary segment can be aligned to the rhythm of the primary segment in various ways.

Path

Source: (SDK Root)\Sample\Multimedia\DirectMusic\PlayMotif

Executable: (SDK Root)\Sample\Multimedia\DirectMusic\Bin

User's Guide

Play the default segment, or load another DirectMusic segment based on a style that contains motifs. Select one of the patterns in the list box and one of the **Align Option** buttons, and then click **Play Motif**. Note how the motif does not begin playing until an appropriate boundary in the primary segment has been reached.

Programming Notes

The PlayMotif sample is very similar in form to the PlayAudio sample. For detailed programming notes on the basics of this sample, see PlayAudio.

When loading the file, PlayMotif performs the same steps as PlayAudio, with the following additions in the LoadSegmentFile function:

1. Loops through styles in the segment, searching it for motifs. It calls **IDirectMusicSegment8::GetParam**, passing GUID_IDirectMusicStyle and an incrementing index to get each of the styles in turn. The method fails when there are no more styles.
2. For each style, it calls **IDirectMusicStyle8::EnumMotif**, passing an incrementing motif index. This call retrieves the motif name at that index. When the call returns S_FALSE, there are no more motifs in the style.

Passes the motif name to **IDirectMusicStyle8::GetMotif** to get an **IDirectMusicSegment8** interface pointer to the motif, and stores this as data associated with the item in the list box.

When **Play Motif** is clicked, the OnPlayMotif function performs the following tasks:

1. Retrieves the desired alignment option from the interface.

2. Gets the selected motif from the listbox, along with its MOTIF_NODE item data. The MOTIF_NODE structure keeps a count of the number of plays currently occurring, as well as a pointer to the **IDirectMusicSegment** interface of the motif.
3. Calls **IDirectMusicPerformance8::PlaySegment**, passing the motif's **IDirectMusicSegment** and flags including DMUS_SEGF_SECONDARY as well as any alignment option.

DirectMusic notifications are handled much as in PlayAudio, but this application also takes note of any motif starting or stopping and updates the play count in the MOTIF_NODE structure. If the play count is greater than zero then it updates the user interface to show that the motif is playing.

See Also

Using Motifs

DirectSound C++ Samples

The following executable applications using the DirectSound API are found on the **Start** menu under **Programs/Microsoft DirectX8 SDK/DirectSound Samples:**

- AdjustSound
- AmplitudeModulation
- CaptureSound
- EnumDevices
- FullDuplexFilter
- Play3DSound
- PlaySound
- SoundFX
- StreamData
- VoiceManagement

The source code is in the following folder:

```
\mssdk\samples\Multimedia\DirectSound
```

In addition to these samples, the source files for tutorial applications are contained in the following folder:

```
\mssdk\samples\Multimedia\DirectSound\Tutorials
```

For more information, see DirectX Audio C++ Tutorials.

The samples other than the tutorials use common source files that implement functions and classes for basic DirectMusic and DirectSound functionality and for

general tasks such as finding media files. These files are found in the following folder:

```
\mssdk\samples\Multimedia\Common\src
```

Note
If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "mssdk" in all paths.

AdjustSound

Description

The AdjustSound sample shows how to create a DirectSound secondary buffer to play a wave file, and how to change the parameters of the buffer.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\AdjustSound

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

Load a wave file by clicking **Sound File**. Select **Focus** and **Buffer Mixing** options; note that the various settings are explained under **Expected Behavior** as you select them. Click **Play**. If you don't hear any sound, check the **Status** pane. The application might fail to create the buffer in hardware if this option has been selected.

By using the sliders you can adjust the frequency, pan, and volume dynamically as the buffer is playing.

Programming Notes

For a simpler example of how to set up a DirectSound buffer without as many controls, see the PlaySound sample.

To set the focus behavior of the buffer, the application calls **IDirectSound8::CreateSoundBuffer** with DSBCAPS_GLOBALFOCUS or DSBCAPS_STICKYFOCUS, or neither of these flags. To set the memory location of the buffer, it sets either DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE, or neither of these flags to allow DirectSound to create the buffer in a either hardware or software.

Parameters are set by using **IDirectSoundBuffer8::SetFrequency**, **IDirectSoundBuffer8::SetPan**, and **IDirectSoundBuffer8::SetVolume**.

See Also

DirectSound Buffers

AmplitudeModulation

Description

The AmplitudeModulation sample shows how to apply an effect to a DirectSound secondary buffer and modify the parameters of the effect.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\AmplitudeModulation

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

Play the default sound or load another wave file by clicking **Sound File**. Change the parameters of the effect by selecting one of the **Wave Form** options and moving the slider to change the modulation rate.

Programming Notes

The application takes the following steps to set an effect on a buffer and change its parameters:

1. Creates the buffer with the DSBCAPS_CTRLFX flag.
2. Describe the effect in a **DSEFFECTDESC** structure, setting the **guidDSFXClass** member to the GUID of the effect desired.
3. Passes the buffer description to **IDirectSoundBuffer8::SetFX**.
4. Calls **IDirectSoundBuffer8::GetObjectInPath** to get an interface pointer to the effect, in this case **IDirectSoundFXGargle8**.
5. In the OnEffectChanged function, uses **IDirectSoundFXGargle8::SetAllParameters** to change the parameters of the gargle effect.

See Also

Using Effects

CaptureSound

Description

The CaptureSound application shows how to capture wave sounds to a file.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\CaptureSound

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

Select a device and a format. Create a file, or replace an existing one, by clicking **Sound File**. Click **Record** to start and stop recording.

Programming Notes

See Readme.txt in the source code folder.

See Also

Capturing Waves

EnumDevices

Description

The EnumDevices sample shows how to enumerate and create playback and capture devices.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\EnumDevices

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

Select a playback and capture device from the dropdown lists. Click **Create**.

Programming Notes

See Readme.txt in the source code folder.

See Also

Enumeration of Sound Devices, Enumeration of Capture Devices

FullDuplexFilter

Description

The FullDuplexFilter application shows how to capture and play back sounds at the same time.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\FullDuplexFilter

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

Be sure your sound card is configured for full-duplex operation and that you have a microphone attached. Choose a format from the **Output Format** list. A list of formats appears in the **Input Format** list. Select one. Click **OK**. If full duplex is successfully created, another dialog box is displayed with a **Record** button. Click this button. Speak into the microphone and your voice is heard over the speakers.

Programming Notes

See Readme.txt in the source code folder.

Play3DSound

Description

The Play3DSound sample shows how to create a 3-D sound buffer and manipulate its properties. It is similar to the 3DAudio sample but uses only the DirectSound API.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\Play3DSound

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

Click **Sound File** and load a wave file. Play the sound. The position of the sound source is shown as a red dot on the graph, where the x-axis is from left to right and

the z-axis is from bottom to top. Change the range of movement on the two axes by using the sliders.

The listener is located at the center of the graph, and has its default orientation, looking along the positive z-axis; that is, toward the top of the screen. The sound source moves to the listener's left and right and to the listener's front and rear, but does not move above and below the listener.

The sliders in the center of the window control the properties of the listener; that is, the global sound properties. If you click **Defer Settings**, changes are not applied until you click **Apply Settings**.

Programming Notes

For a simpler example of how to set up a DirectSound buffer without 3-D positioning, see the PlaySound sample.

The sample obtains an **IDirectSound3DListener8** interface as follows:

1. Describes a buffer in a **DSBUFFERDESC** structure containing the DSBCAPS_CTRL3D and DSBCAPS_PRIMARYBUFFER flags.

2. Passes the buffer description to **IDirectSound8::CreateSoundBuffer**. This creates a primary buffer object with 3-D capabilities.

3. Calls **QueryInterface** to obtain the **IDirectSound3DListener8** interface. This interface controls global 3-D properties.

The sample obtains an **IDirectSound3DBuffer8** interface as follows:

1. Describes a buffer in a **DSBUFFERDESC** structure, setting the DSBCAPS_CTRL3D capabilties flag and and a 3-D virtualization algorithm.
2. Passes the buffer description to **IDirectSound8::CreateSoundBuffer**, creating a secondary buffer object with 3-D capabilities.
3. Calls **QueryInterface** to obtain the **IDirectSound3DBuffer8** interface. This interface controls the 3-D properties of sounds played through the buffer.

The application sets the parameters of the listener by calling **IDirectSound3DListener8::SetAllParameters**. The listener properties are described in a **DS3DLISTENER** structure. To set the position of the sound source, the application calls **IDirectSound3DBuffer8::SetAllParameters**, passing in a **DS3DBUFFER** structure.

If the DS3D_DEFERRED flag is set when parameters are set, nothing is actually changed until **IDirectSound3DListener8::CommitDeferredSettings** is called. This method commits all deferred settings for buffers as well as the listener, and makes it possible for DirectSound to perform calculations once instead of many times.

See Also

3-D Sound

PlaySound

Description

The PlaySound sample shows how to play a wave file in a DirectSound secondary buffer.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\PlaySound

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

Load a wave file by clicking **Sound File**. Select **Loop Sound** if you want it to play repeatedly. Click **Play**.

Programming Notes

The sample shows the basic tasks required to play a sound in a DirectSound Buffer, as follows:

Set up DirectSound:

1. Call the **DirectSoundCreate** function to create the DirectSound object.
2. Call **IDirectSound8::SetCooperativeLevel**.
3. Set the primary buffer format. This sample calls the DSUtil_SetPrimaryBufferFormat function in Dsutil.cpp to do this.

Load a wave file into a DirectSound buffer:

1. Read the wave file header to get the data size and format.
2. If the wave file is a reasonable size, create a secondary buffer in the same format as the data, large enough to hold the entire file. If the wave file is too large, data will have to be streamed to the buffer. See the StreamData sample.
3. Fill the buffer with data. Obtain a pointer into the buffer by calling **IDirectSoundBuffer8::Lock**. Because data is not being streamed as the buffer is playing, the entire buffer can be locked. After the memory has been copied, call **IDirectSoundBuffer8::Unlock**.

Play the DirectSound buffer:

1. Check to see if the buffer has been lost. If so, restore it.
2. To play the buffer call **IDirectSoundBuffer8::Play**.

Free DirectSound:

Call **Release** on all the DirectSound interfaces that were obtained. Release buffers before releasing the DirectSound object.

See Also

Filling and Playing Static Buffers

SoundFX

Description

The SoundFX sample shows how to use DMOs with DirectSound and how to manipulate effect parameters. It is similar to the AudioFX sample but does not use an audiopath or any of the DirectMusic interfaces.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\SoundFX

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

A default sound file is loaded when the application is run. If you wish, load a different one by clicking **Open File**.

At first, no effects are enabled. Click **Play** to hear the sound without effects.

Click **Stop** to stop the buffer. Apply one or more effects by selecting checkboxes in the **Enable** column. Play the sound again.

To adjust parameters for an effect, select an option button in the **Adjust** column and change the values in the frame on the right side of the window. This can be done regardless of whether the sound is playing and regardless of whether the effect has been applied yet.

Programming Notes

The application implements a CSoundFXManager class to manage effects. An **IDirectSoundBuffer8** interface is used to set effects on the buffer in the CSoundFXManager::ActivateFX method. Effect parameters are set in the OnEffectChanged function in response to messages from the interface.

See Also

Using Effects, Using Effects in DirectSound

StreamData

Description

The StreamData sample shows how to stream a wave file to a DirectSound secondary buffer. It is similar to the PlaySound sample, but adds support for streaming.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\StreamData

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

Load a wave file by clicking **Sound File**. Select **Loop Sound** if you want it to play repeatedly. Click **Play**.

Programming Notes

The sample shows the basic tasks required to play a streaming sound in a DirectSound secondary buffer.

Set up DirectSound:

1. Call the **DirectSoundCreate8** function to create the DirectSound object.
2. Call **IDirectSound8::SetCooperativeLevel**.
3. Set the primary buffer format. The sample calls the DSUtil_SetPrimaryBufferFormat function in Dsutil.cpp to do this.

Create a DirectSound buffer and set up the notifications:

1. Read the wave file header to get the data size and format.
2. Choose a DirectSound buffer size. For this sample, the buffer holds about 3 seconds of data.
3. Create a DirectSound buffer of that size and with the same format as the wave file. Also set the DSBCAPS_CTRLPOSITIONNOTIFY flag so that the buffer can send notification events whenever playback has reached certain points in the buffer. On some drivers, using this flag limits the buffer to software, because the hardware buffers do not support position notifications.
4. Set up the notifications on the buffer by obtaining an **IDirectSoundNotify8** interface from the buffer and calling **IDirectSoundNotify8::SetNotificationPositions**. See the InitDSoundNotification function. When the play cursor passes a notification position, it signals a Win32® event.

Play the DirectSound buffer:

1. See if the buffer has been lost, and call **IDirectSoundBuffer8::Restore** if so.
2. Fill the entire buffer with data from the file.
3. Call **IDirectSoundBuffer8::Play** with the DSBPLAY_LOOPING flag. The looping flag needs to be set so that the buffer will continue playing after the first batch of data has been played.

Check to see if a notification is signaled:

1. Look for a signaled event in the message loop or in a separate thread by calling **MsgWaitForMultipleObjects**.
2. If a buffer notification event has been signaled, lock the section of the buffer that has just been played and fill it with new data. See the HandleNotification function.

Stop the buffer:

When handling the event notification, keep track of how much data has been put in the buffer. When the entire wave file has been put into the buffer, and after DirectSound has played it all, it is necessary to manually stop the buffer since the buffer will continuously loop otherwise.

See Also

Using Streaming Buffers

VoiceManagement

Description

The VoiceManagement sample shows how to implement dynamic voice management when creating DirectSound secondary buffers.

Path

Source: (SDK Root)\Sample\Multimedia\DirectSound\VoiceManagement

Executable: (SDK Root)\Sample\Multimedia\DirectSound\Bin

User's Guide

Load a wave file by clicking **Sound File**. Select the **Voice Allocation Flags**, **Buffer Priority**, and **Voice Management Flags** options. Note that the effect of the selected options is described under **Expected Behavior**. Create the buffer and play the sound by clicking **Play**.

Programming Notes

To use voice management flags, the buffer must be created using DSBCAPS_LOCDEFER. Otherwise DirectSound will not be able to dynamically place the buffer in either hardware or software at run time.

When playing the buffer, call **IDirectSoundBuffer8::Play** with a valid combination of voice management flags. Run the sample to observe the result of any combination of flags.

See Also

Dynamic Voice Management

DirectX Audio Visual Basic Tutorials

This section contains the following tutorials showing how to implement Microsoft® DirectMusic® and Microsoft® DirectSound® in a Visual Basic® application:

- Tutorial 1: Playing Audio Files
- Tutorial 2: Using Audiopaths

Other, more specialized uses of Microsoft® DirectX® Audio are demonstrated in the sample applications provided with the SDK. For a description of these samples, see DirectX Audio Visual Basic Samples.

Tutorial 1: Playing Audio Files

This tutorial is a guide to performing the most basic tasks in DirectX Audio: initializing a DirectMusic performance and playing an audio file. The tutorial is presented in the following steps:

- Step 1: Set Up the Audio System
- Step 2: Load a File
- Step 3: Play the File
- Step 4: Close Down

The complete sample code for the tutorial is available in the following folder:

```
\mssdk\samples\Multimedia\VBSamples\DirectMusic\Tutorials\Tut1
```

Note

If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "mssdk" in the path.

Step 1: Set Up the Audio System

The sample application declares the following object variables:

```
Private dx As New DirectX8
Private dml As DirectMusicLoader8
Private dmp As DirectMusicPerformance8
Private seg As DirectMusicSegment8
```

The only one of these objects that can be initialized during declaration is the **DirectX8** object. **DirectMusicLoader8** and **DirectMusicPerformance8** must be created by using methods of this object, as follows:

```
Set dml = dx.DirectMusicLoaderCreate
Set dmp = dx.DirectMusicPerformanceCreate
```

After the performance has been created, it must be initialized. The **DirectMusicPerformance8.InitAudio** method performs the following tasks:

- Creates a **DirectSound8** object. In most cases you don't need a variable for this object, and you can pass Nothing as the *DirectSound* parameter, or omit it.
- Associates an application window with the DirectSound object. Normally, the handle of the main application window is passed as the *hwnd* parameter.
- Sets up a default audiopath of a standard type. The tutorial requests a path of type DMUS_APATH_SHARED_STEREOPLUSREVERB, which is suitable for music.
- Allocates a number of performance channels to the audiopath. Wave files require only a single performance channel, and MIDI files require up to 16. Segments created in DirectMusic Producer might need more. No harm is done by asking for extra channels.
- Specifies capabilities and resources of the synthesizer. This can be done in one of two ways: by setting flags or by supplying more detailed information in the **DMUS_AUDIOPARAMS** type. Most applications set the DMUS_AUDIOF_ALL flag and let DirectMusic create the synthesizer with default parameters. A **DMUS_AUDIOPARAMS** type still has to be passed to the method, but it can be left blank, as in the **DMUS_AUDIOPARAMS** typesample.

In the sample application, the call to **InitAudio** is very simple:

```
dmp.InitAudio Me.hWnd, DMUS_AUDIOF_ALL, dmA, Nothing, _
    DMUS_APATH_SHARED_STEREOPLUSREVERB, 64
```

Next: Step 2: Load a File

Step 2: Load a File

The **DirectMusicLoader8** object created in the preceding step can be used throughout the lifetime of the application to load sound data from files.

To get the data, the loader needs to know where to find it. Although a full path can be provided each time a file is loaded, it is more convenient to establish a default directory. Do this by using the **DirectMusicLoader8.SetSearchDirectory** method. The sample application uses the path returned by its FindMediaDir function. This function finds the directory where the sample file is located. The loader could subsequently load any file from this directory without being given more than the name.

```
dml.SetSearchDirectory FindMediaDir("sample.sgt", True)
```

The loader then loads the file and returns it as a **DirectMusicSegment8** object:

```
Set seg = dml.LoadSegment("sample.sgt")
```

Next: Step 3: Play the File

Step 3: Play the File

The file loaded in the previous step is now available to the performance as a **DirectMusicSegment8** object.

Before a segment loaded from a MIDI file or a DirectMusic Producer segment file can be played, its band must be downloaded to the synthesizer. As long as you don't unload the band, this step has to be taken only once for each segment that uses a unique band.

The following code from the sample application downloads the band to the default audiopath. Alternatively, it could be downloaded to the performance. As long as only a single synthesizer is in use, it doesn't matter which destination object you choose.

```
seg.Download dmp.GetDefaultAudioPath
```

To play the file, pass the segment object to **DirectMusicPerformance8.PlaySegmentEx**. This method offers many options for playback, but to play a segment immediately on the default audiopath, all the parameters except the first can be 0 or omitted, as in the following code from the sample application:

```
dmp.PlaySegmentEx seg, 0, 0
```

Next: Step 4: Close Down

Step 4: Close Down

To exit an audio application cleanly, you must perform three main steps:

- Stop any playing segments by calling **DirectMusicPerformance8.StopEx**.
- Close down the performance. The **DirectMusicPerformance8.CloseDown** method performs miscellaneous cleanup tasks and releases internal references to objects.
- Release all objects by setting object variables to Nothing.

The following procedure from the sample application closes down DirectX Audio:

```
Private Sub Form_Unload(Cancel As Integer)
    If Not (seg Is Nothing) Then
        dmp.StopEx dmp.GetDefaultAudioPath, 0, 0
    End If
    Set seg = Nothing
    dmp.CloseDown
    Set dmp = Nothing
    Set dml = Nothing
    Set dx = Nothing
End Sub
```

Tutorial 2:: Using Audiopaths

This tutorial shows how to set up a DirectX Audio application that uses an audiopath other than the default audiopath. It also demonstrates how to set the volume on the audiopath and obtain an object from the audiopath to set parameters. In this case, the object obtained is a **DirectSound3DBuffer8** object, which can be used to locate the sound in space.

The tutorial is presented in the following steps:

- Step1: Create the DirectX Objects
- Step 2: Initialize the Audio Environment
- Step 3: Create the Audiopath
- Step 4: Load a Segment
- Step 5: Download the Band
- Step 6: Play the Sound on the Audiopath
- Step 7: Stop the Sound
- Step 8: Set Audiopath Parameters
- Step 9: Set 3-D Parameters

The complete sample code for the tutorial is available in the following folder:

```
\mssdk\samples\Multimedia\VBSamples\DirectMusic\Tutorials\Tut2
```

Note

If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "mssdk" in the path.

Step1: Create the DirectX Objects

The first step is to create the DirectX8 object, which is then used to create DirectMusicLoader8 and DirectMusicPerformance8.

The variables for these objects are declared as follows:

```
Private dx As DirectX8
Private dml As DirectMusicLoader8
Private dmp As DirectMusicPerformance8
```

In addition, the following global object variables are declared for later use:

```
Private dmSeg As DirectMusicSegment8
Private dmSegState As DirectMusicSegmentState8
Private dmPath As DirectMusicAudioPath8
```

The first three objects are created in the Form_Load procedure. If any errors occur, they are ignored until after each creation method has been tried. If any error occurred, a single message box is displayed and the application terminates.

```
Private Sub Form_Load()

    Dim dmA As DMUS_AUDIOPARAMS
    MediaPath = FindMediaDir("tuta.wav")

    On Local Error Resume Next

    Set dx = New DirectX8
    Set dml = dx.DirectMusicLoaderCreate
    Set dmp = dx.DirectMusicPerformanceCreate
```

The Form_Load procedure is continued in the next topic, Step 2: Initialize the Audio Environment

Step 2: Initialize the Audio Environment

Still in the Form_Load procedure, the sample application sets the search directory so the loader knows where to find the sound files:

```
    dml.SetSearchDirectory MediaPath
```

Next it initializes the performance. The **DMUS_AUDIOPARAMS** type is left blank, causing the performance to be initialized with default parameters. No default audiopath is requested.

```
dmp.InitAudio Me.hWnd, DMUS_AUDIOF_ALL, dmA
```

If any errors have occurred in creating or initializing the objects, they are handled now:

```
If Err.Number <> 0 Then
   MsgBox "Could not initialize DirectMusic." & vbCrLf & _
      "This sample will exit.", vbOKOnly Or vbInformation, _
      "Exiting..."
   Unload Me
   Exit Sub
End If
```

The Form_Load procedure is concluded in the next topic, Step 3: Create the Audiopath

Step 3: Create the Audiopath

Recall that in the call to **DirectMusicPerformance8.InitAudio**, no default audiopath was specified. Instead, the sample creates a standard audiopath on which segments can be played. In a more complex application, multiple audiopaths might be created: one for music and one for 3-D sound effects, for example. This tutorial requires only a single audiopath, of type DMUS_APATH_DYNAMIC_3D. This type of audiopath does not contain any effects such as reverberation, but is adequate for simple sound effects.

```
   Set dmPath = dmp.CreateStandardAudioPath(DMUS_APATH_DYNAMIC_3D, _
                                64, True)
End Sub  ' Form_Load
```

The audiopath contains support for up to 64 performance channels and is activated on creation.

Next: Step 4: Load a Segment

Step 4: Load a Segment

The application now has all the objects it needs for loading and playing sounds. In the sample, one of three sound files is always played in response to the **Play** button, depending on the user's choice of file type.

The LoadSound procedure is called in response to the first click of the **Play** button, and also in response to the selection of an option button. The first thing the procedure does is verify that the file exists, as follows:

```
If Dir$(sFile) = vbNullString Then
   MsgBox "Unable to find " & sFile, vbOKOnly Or vbCritical, _
        "Not found."
```

```
    Unload Me
  End If
```

If a segment has already been loaded, its band or bands are unloaded from the synthesizer, and the segment is released:

```
  If Not (dmSeg Is Nothing) Then
    dmSeg.Unload dmPath
    Set dmSeg = Nothing
  End If
```

Then the new segment is loaded. If it is a MIDI file, the application calls **DirectMusicSegment8.SetStandardMidiFile**. A standard MIDI file behaves somewhat differently than one that has been saved as a DirectMusic Producer segment, and this call ensures that it is handled properly.

```
  Set dmSeg = dml.LoadSegment(sFile)
  If Right$(sFile, 4) = ".mid" Then dmSeg.SetStandardMidiFile
```

The LoadSound procedure is concluded in the next topic, Step 5: Download the Band

Step 5: Download the Band

An easily overlooked step in playing a segment is to download the band. This step is not needed for wave files, but does no harm. For MIDI files and DirectMusic Producer segments, it is essential, so that the synthesizer can play note messages with the correct timbres.

The **DirectMusicSegment8.Download** method takes either an audiopath or the performance as a parameter. Since most applications use only a single synthesizer, it doesn't matter which you use. In the sample, the segment's bands are downloaded to the audiopath created in Step 3: Create the Audiopath.

```
dmSeg.Download dmPath
```

The LoadSound procedure concludes by initializing the volume and 3-D positioning of the sound:

```
    scrlPan_Change
    scrlVol_Change
End Sub
```

Next: Step 6: Play the Sound on the Audiopath

Step 6: Play the Sound on the Audiopath

When the **Play** button is clicked, the following procedure checks the state of the option buttons, loads the appropriate segment if one has not already been loaded in

response to an option button selection, and plays the segment as a primary segment on the audiopath created previously.

```
Private Sub cmdPlay_Click()

    If dmSeg Is Nothing Then
        If optWave.Value Then LoadSound MediaPath & "\tuta.wav"
        If optMid.Value Then LoadSound MediaPath & "\tut.mid"
        If optSeg.Value Then LoadSound MediaPath & "\sample.sgt"
    End If

    Set dmSegState = dmp.PlaySegmentEx(dmSeg, 0, 0, Nothing, dmPath)

End Sub
```

The procedure obtains a **DirectMusicSegmentState8** object, which could be used to retrieve information about the state of the playback. The segment state can also be passed to **DirectMusicPerformance8.StopEx** to stop just one instance of the segment. This is useful for secondary segments, where multiple instances might be playing at the same time.

Next: Step 7: Stop the Sound

Step 7: Stop the Sound

The following procedure stops the sound in response to input. Because no repetitions are set for any of the sample segments, they also stop automatically when they reach the end.

```
Private Sub cmdStop_Click()
    If dmSeg Is Nothing Then Exit Sub
    dmp.StopEx dmSeg, 0, 0
    mlOffset = 0
End Sub
```

The call to **DirectMusicPerformance8.StopEx** takes the segment as an argument; it might also take the segment state obtained in the previous step, or the audiopath. The second parameter orders an immediate stop, and the third specifies no flags. The flags could request that the music not stop until an appropriate boundary occurs, such as the end of a measure, but such a request would be ignored for wave files.

Next: Step 8: Set Audiopath Parameters

Step 8: Set Audiopath Parameters

At any time, the user of the sample application can set the volume on the audiopath by moving the slider. In response, the application calls **DirectMusicAudioPath8.SetVolume**. The change affects all segments played on the

audiopath. If the application had multiple audiopaths, segments played on other audiopaths would not be affected.

```
Private Sub scrlVol_Change()
    dmPath.SetVolume scrlVol.Value, 0
End Sub
```

The first parameter is the new volume, in hundredths of a decibel. The application allows the volume to be lowered by up to 50 decibels. The second value is the fade time, or the duration over which the change in volume takes place. In this case, the new volume is set immediately.

Next: Step 9: Set 3-D Parameters

Step 9: Set 3-D Parameters

In this step you learn how to obtain the **DirectSound3DBuffer8** object from the audiopath and use it to locate the sound in space. Although the sample application only pans the sound from right to left, the same technique can be used to move the sound in three dimensions.

In Step 3: Create the Audiopath, when the performance was initialized, the application requested a default audiopath of type DMUS_APATH_DYNAMIC_3D. This standard audiopath sends its data through a 3-D buffer. When the user moves the pan slider, the application responds first by obtaining an object for this buffer.

```
Private Sub scrlPan_Change()
    If dmSeg Is Nothing Then Exit Sub

    Dim dsBuf As DirectSound3DBuffer8

    Set dsBuf = dmPath.GetObjectinPath(DMUS_PCHANNEL_ALL, _
        DMUS_PATH_BUFFER, 0, vbNullString, 0, _
        "IID_IDirectSound3DBuffer")
```

The parameters to **DirectMusicAudioPath8.GetObjectInPath** can be a little tricky to set up properly. For information on which parameters are relevant for objects at different stages in the path, see Retrieving Objects from an Audiopath.

In this case, you are retrieving a secondary buffer that is used by all performance channels on this audiopath. Set the *lPChannel* parameter to DMUS_PCHANNEL_ALL.

Because the buffer you want is the first of the two in the chain, you pass 0 as *lBuffer*. The DMUS_PATH_BUFFER stage contains only buffer objects, and not the DMOs attached to those buffers; therefore *lIndex* is ignored.

Now the application can use the buffer object to set the position of the sound source, as follows:

```
    dsBuf.SetPosition scrlPan.Value / 5, 0, 0, DS3D_IMMEDIATE
    Set dsBuf = Nothing

End Sub
```

The first three parameters specify the new position of the sound source in relation to the default listener. The default listener is at coordinates (0.0, 0.0, 0.0), facing toward the positive z-axis, with the top of the head toward the positive y-axis. Distance units are meters by default. For more information, see Coordinates of 3-D Space and Listener Orientation.

The last parameter of the **DirectSound3DBuffer8.SetPosition** method specifies whether the change is to be made immediately or deferred until all changes are committed. For more information, see Deferred Settings.

DirectX Audio Visual Basic Samples

The sample applications described in this section demonstrate the use and capabilities of the Microsoft® DirectMusic® and Microsoft® DirectSound® application programming interfaces (APIs) in Microsoft® DirectX® for Visual Basic®.

The samples are described under the following headings:

- DirectMusic Visual Basic Samples. These samples use the DirectMusic API to load and play sounds. Some of them also use elements of the DirectSound API.
- DirectSound Visual Basic Samples. These samples use only the DirectSound API. However, many of the techniques shown can also be used with buffers obtained from a DirectMusic audiopath.

DirectMusic Visual Basic Samples

The following executable applications using the DirectMusic API, or DirectSound and DirectMusic together, are found on the **Start** menu under **Programs/Microsoft DirectX8 SDK/Visual Basic Samples/DirectMusic Samples**:

- AudioEffects
- AudioPath
- DLSEffects
- DMDrums
- PlayAudio

- PlayMotif
- SimpleAudioPath

The source code is in the following folder:

```
\mssdk\samples\Multimedia\VBSamples\DirectMusic\
```

In addition to these samples, the source files for tutorial applications are contained in the following folder:

```
\mssdk\samples\Multimedia\VBSamples\DirectMusic\Tutorials
```

For more information, see DirectX Audio Visual Basic Tutorials.

Note

If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "mssdk" in all paths.

AudioEffects

Description

The AudioEffects sample shows how to retrieve a buffer from an audiopath and apply effects to it. It also demonstrates the standard effects supplied with DirectX.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\AudioEffects

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\Bin

User's Guide

Load a sound file. Add one or more effects to the **Effects in Use** list by selecting an effect from the **Available Effects** list and clicking the right arrow. Remove an effect by selecting it in the **Effects in Use** list and clicking the left arrow. To update the effects on the buffer, click **Apply Effects**.

Programming Notes

The cmdApply_Click procedure contains the code for setting the effects, or removing them. Note the main steps:

- Describe the effects in the DSEffects array of **DSEFFECTDESC** types.
- Obtain the buffer from the audiopath by calling **DirectMusicAudioPath8.GetObjectInPath**.
- Deactivate the audiopath.
- Set the effects by calling **DirectSoundSecondaryBuffer8.SetFX**.

- Reactivate the audiopath.

AudioPath

Description

The AudioPath sample shows how files can be played on separate audiopaths, each of which can be manipulated separately.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\AudioPath

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\Bin

User's Guide

Load one file on Audiopath 1 and one or more files on Audiopaths 2 and Audiopath 3. Click **Play** for any audiopath to play all files listed for that path. You can play more than one audiopath at the same time. Use the sliders to change the volume and 3-D spatialization on each audiopath.

Programming Notes

In the InitAudio procedure, the performance is initialized without a default audiopath, and three standard audiopaths are created. Each segment is played on a particular audiopath by passing that audiopath to **DirectMusicPerformance8.PlaySegmentEx**.

It is possible to play multiple instances of a segment on a single audiopath by loading the file more than once as a secondary segment. Because the instances play simultaneously in the sample, you only hear one instance. However, the sample does illustrate how each instance is represented by a **DirectMusicSegmentState8** object, which is passed to **DirectMusicPerformance8.IsPlaying** to ascertain whether the instance is playing.

DLSEffects

Description

The DLSEffects sample shows how an application can use DLS instruments for sound effects, and how to send MIDI messages.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\DLSEffects

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\Bin

User's Guide

Click the buttons labeled **C3** to **C10** to play a vocal effect. Click **On** to play the heartbeat effect continuously, and vary the pitch by using the **Note** and **Pitch Bend** sliders. Set the velocity (volume) of notes by using the slider on the right. Changes in velocity apply only to sounds started after the change is made.

Programming Notes

The DLS instruments are taken from Boids.dls. That collection contains only a single instrument, called Vocals, which is based on different wave samples for different regions, or ranges of notes. For example, the first speech sound is used when any MIDI note between C3 and B3 is played. The speech sounds are played at the original pitch only when the note is the lowest one in the region.

One of the samples, called Heartbeat, is valid for the range B7-B8. Because this is not a speech sample, the pitch can reasonably be varied. This is done by playing various notes within the range, as determined by the slider setting.

Heartbeat is also the only sample in the DLS collection that is based on a looped wave, so that it can be played continuously as long as the note is on. The other samples play only once regardless of the duration of the note.

DMDrums

Description

The DMDrums sample application shows how an application can vary music by changing the band, volume, master groove level, and tempo. It also shows how to play motifs in response to user input.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\DMDrums

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\Bin

User's Guide

Start the music by clicking the **Play** button. Select different types of music from the **Groove** list and different bands from the **Drum Sets** list. Adjust tempo and volume with the UpDown controls. Play drum motifs by clicking on any of the drum buttons.

Programming Notes

The selection in the **Groove** list determines the master groove level, which causes different patterns to be selected from the style object associated with the segment.

Every drum button plays a different motif. The instrument associated with the button is assigned in the motif.

PlayAudio

Description

The PlayAudio sample plays a sound on an audiopath and enables the user to set the tempo and master volume.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\PlayAudio

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\Bin

User's Guide

Load a segment, MIDI file, or wave by clicking **Audio File**. Select playback options. The **Tempo** slider is not enabled for wave files.

Programming Notes

The application sets up a notification of type DMUS_NOTIFY_ON_SEGMENT, which triggers the DirectXEvent8_DXCallback procedure on various segment-related events. In the callback procedure, the subtype of the notification message is checked to see if the segment has stopped playing. If it has, the user interface is updated to enable play options.

PlayMotif

Description

The PlayMotif sample demonstrates how a motif played as a secondary segment can be aligned to the rhythm of the primary segment in various ways.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\PlayMotif

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\Bin

User's Guide

Play the default segment, or load another DirectMusic Producer segment that contains motifs. Select one of the patterns in the list box and one of the **Align Option** buttons,

and then click **Play Motif**. Note how the motif does not begin playing until an appropriate boundary in the primary segment has been reached.

Programming Notes

Information about the motifs is contained in an array of user-defined types, which is initialized in the LoadSegment procedure. When a motif is played, it is aligned to the primary segment's rhythm by passing one of the **CONST_DMUS_SEGF_FLAGS** to **DirectMusicPerformance8.PlaySegmentEx**, in the cmdPlayMotif_Click procedure.

Notifications are used to keep the user interface up to date. Of particular interest is the way the segment state is retrieved from the **User** member of the **DMUS_NOTIFICATION_PMSG** type, so that the application can ascertain which motif started or stopped.

SimpleAudioPath

Description

The SimpleAudioPath sample demonstrates how different sounds can be played on an audiopath, and how the parameters of all sounds are affected by changes made on the audiopath.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\SimpleAudioPath

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectMusic\Bin

User's Guide

Click **Lullaby**, **Snore**, and **Mumble** to play different sounds. Adjust the 3-D position of the sounds by using the sliders. Click **Rude Awakening** to stop all other sounds.

Programming Notes

The lullaby music is played as the primary segment, which means that only one instance of it can be playing at one time. You can play multiple instances of the snore and mumble sounds.

The audiopath is of type DMUS_APATH_DYNAMIC_3D. The path contains a 3-D buffer into which the synthesizer mixes all the sounds. To change the position of the sound, the application retrieves a **DirectSound3DBuffer8** object from the path and calls **DirectSound3DBuffer8.SetPosition**.

When **Rude Awakening** is clicked, the audiopath is passed to **DirectMusicPerformance8.PlaySegmentEx** as the *From* parameter. This causes all segment instances playing on the audiopath to stop before the new segment begins.

DirectSound Visual Basic Samples

The following executable applications using only DirectSound are found on the **Start** menu under **Programs/Microsoft DirectX8 SDK/Visual Basic Samples/DirectSound Samples**:

- AdjustSound
- DeferredEffects
- EffectsBuffers
- EnumDevice
- Play3DSound

The source code for these samples is in the following folder:

```
\mssdk\samples\Multimedia\VBSamples\DirectSound\
```

In addition to these samples, the source files for the tutorial applications are contained in the following folder:

```
\mssdk\samples\Multimedia\VBSamples\DirectSound\Tutorials
```

For more information, see DirectX Audio Visual Basic Tutorials.

Note

If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "mssdk" in all paths.

AdjustSound

Description

The AdjustSound sample shows how to create a DirectSound secondary buffer to play a wave file, and how to change the parameters of the buffer.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\AdjustSound

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\Bin

User's Guide

Load a wave file by clicking **Sound File**. Select the **Focus** and **Buffer Mixing** options; note that the various settings are explained under **Expected Behavior** as you select them. Click **Play**. If you don't hear any sound, check the **Status** pane. The application might fail to create the buffer in hardware if this option has been selected.

By using the sliders, you can adjust the frequency, pan, and volume dynamically as the buffer is playing.

Programming Notes

The application attempts to create a sound buffer when the file is loaded, and each time **Play** is clicked, using whatever options are selected at the time.

DeferredEffects

Description

The DefferedEffects sample demonstrates how to create deferred sound buffers with effects. Resources are not allocated to the buffers until the application requests them.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\DeferredEffects

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\Bin

User's Guide

Load a wave file by clicking the button beside the top text box. Select an effect in the top list and move it to the bottom list by clicking the down arrow, or by double-clicking the effect. Remove an effect by selecting it in the bottom list and clicking on the up arrow, or by double-clicking the effect. Update the effects on the buffer by clicking **Apply Effects**. Note that the status of each effect is given in the **Effects in Use** list. Play the sound.

Programming Notes

The sound buffer is created with the DSBCAPS_LOCDEFER flag, so that no resources are allocated to it until it is played, or until the application calls **DirectSoundSecondaryBuffer8.AcquireResources**.

The buffer is still deferred when **DirectSoundSecondaryBuffer8.SetFX** is called. The application then immediately calls **AcquireResources** to allocate resources to the buffer. It then ascertains whether the effects were successfully instantiated, and whether they are in hardware or software. (In DirectX 8.0, all effects are in software.)

EffectsBuffers

Description

The EffectsBuffers sample demonstrates the use of effects on DirectSound secondary buffers, and enables modification of effect parameters.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\EffectsBuffers

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\Bin

User's Guide

Load a wave file by clicking the button beside the top text box. Select an effect in the top list and move it to the bottom list by clicking the down arrow, or by double-clicking the effect. Remove an effect by selecting it in the bottom list and clicking on the up arrow, or by double-clicking the effect, or by right-clicking the effect and choosing **Remove** from the popup menu. Update the effects on the buffer by clicking **Apply Effects**. Note that the status of each effect is given in the **Effects in Use** list. Play the sound.

You can modify the parameters for an effect when the sound is not playing. Select the effect in the **Effects In Use** list and click **Modify Effect**, or right-click the effect and choose **Change Settings** from the popup menu.

Programming Notes

The ApplySettings function keeps track of what effects have already been applied by creating a unique value for each combination of effects. When the user clicks **Apply Effects**, the current combination is compared with the last one, stored in *mlEffectKey*, to see whether the effects have to be updated.

EnumDevice

Description

The EnumDevice sample shows how to enumerate and create playback and capture devices.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\EnumDevice

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\Bin

User's Guide

Select a playback and capture device from the dropdown lists. Click **Create**.

Programming Notes

The LoadEnum procedure populates the listboxes. The primary devices on the system are always enumerated twice, once under their proper names and once as "Primary

Sound Driver" or "Primary Sound Capture Driver." The value returned by **DirectSoundEnum8.GetGuid** for the primary driver is all zeros.

The **DirectSound8** object is released as soon as it has been created. On systems that do not support full duplex, the creation of **DirectSoundCapture8** could fail if the **DirectSound8** object exists.

Play3DSound

Description

The Play3DSound sample shows how to create a 3-D sound buffer and manipulate its properties.

Path

Source: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\Play3DSound

Executable: (SDK Root)\Sample\Multimedia\VBSamples\DirectSound\Bin

User's Guide

Load a mono wave file by clicking **Sound**. Choose a 3-D virtualization algorithm. Play the sound, and set the volume and direction of the sound source by using the sliders. The direction is the orientation of the sound cone; the sound cone is oriented toward the narrow point of the red triangle. Move the sound source by dragging the red triangle or by clicking on a destination. The listener is represented by the black triangle, which cannot be moved.

Programming Notes

Because of the limitations of a two-dimensional display, the sound source can move only along the x-axis (positive toward the right of the screen) and along the z-axis (positive toward the bottom of the screen). The application uses the default listener position (0, 0, 0) and orientation (0, 0, 1). This means that the listener is facing the top of the screen.

If the user attempts to load a stereo file, an error results, because a buffer in a stereo wave format cannot be created with the DSBCAPS_CTRL3D flag. The **DSBUFFERDESC** type passed to **DirectSound8.CreateSoundBufferFromFile** receives information about the format even though the method fails, making it possible for the application to warn the user after handling the error. Even if the method succeeds, the buffer is discarded because the application creates a new one after ascertaining the user's preferred virtualization algorithm. A **DirectSound3DBuffer8** object is obtained from this buffer, and this object is used for controlling the 3-D parameters.

DirectMusic C/C++ Reference

This section contains reference information for the API elements of DirectMusic. Reference material is divided into the following categories.

- DirectMusic Interfaces
- DirectMusic Messages
- DirectMusic Structures
- DLS Structures
- DirectMusic File Format
- DirectMusic File Structures
- Standard Track Parameters
- DirectMusic Enumerated Types
- DirectMusic Return Values

DirectMusic Interfaces

This section contains references for DirectMusic COM interfaces.

Interfaces in the current version are either declared or defined with names ending in **8.** To be sure of using the latest version of an interface, always include this suffix when declaring the pointer.

Note

IKsControl and **IReferenceClock** are exceptions to this rule.

Interfaces retrieved by methods are always the base version. Where a newer version exists, you must call **QueryInterface** to obtain it, as in the following example, where *lpdmBand* is an **IDirectMusicBand8** interface pointer.

```
IDirectMusicSegment * lpdmseg;
IDirectMusicSegment8 * lpdmseg8;

HRESULT hr = lpdmBand->CreateSegment(&lpdmseg);
if (SUCCEEDED(hr))
{
   hr = lpdmseg->QueryInterface(IID_IDirectMusicSegment8,
      (LPVOID *)&lpdmseg8);
}
```

Where there is no new version of an interface, the interface name with the suffix **8** is only a define. For example, **IDirectMusicGraph8** is equivalent in all respects to **IDirectMusicGraph**. In such cases it is not necessary to query for a new interface, but doing so does no harm and can make your code easier to maintain for future versions of DirectX.

When a method takes an interface pointer as an IN parameter, you can pass in the newer version even where the method is declared as accepting the older version. For example, a pointer to either **IDirectMusicSegment** or **IDirectMusicSegment8** can be passed to **IDirectMusicPerformance8::PlaySegmentEx**.

This documentation contains full reference topics only for the latest versions of interfaces. Where a define exists, such as **IDirectMusicGraph8**, the interface is documented under that name.

- **IDirectMusic8**
- **IDirectMusicAudioPath8**
- **IDirectMusicBand8**
- **IDirectMusicBuffer8**
- **IDirectMusicChordMap8**
- **IDirectMusicCollection8**
- **IDirectMusicComposer8**
- **IDirectMusicContainer8**
- **IDirectMusicDownload8**
- **IDirectMusicDownloadedInstrument8**
- **IDirectMusicGetLoader8**
- **IDirectMusicGraph8**
- **IDirectMusicInstrument8**
- **IDirectMusicLoader8**
- **IDirectMusicObject8**
- **IDirectMusicPatternTrack8**
- **IDirectMusicPerformance8**
- **IDirectMusicPort8**
- **IDirectMusicPortDownload8**
- **IDirectMusicScript8**
- **IDirectMusicSegment8**
- **IDirectMusicSegmentState8**
- **IDirectMusicSong8**
- **IDirectMusicStyle8**
- **IDirectMusicThru8**
- **IDirectMusicTool8**
- **IDirectMusicTrack8**
- **IKsControl**
- **IReferenceClock**

IDirectMusic8

The **IDirectMusic8** interface provides methods for managing buffers, ports, and the master clock. There should not be more than one instance of this interface per application.

IDirectMusic8 supersedes **IDirectMusic** and adds a new method, **SetExternalMasterClock**.

There is no helper function to create this interface. Applications use the COM **CoCreateInstance** function, the **IDirectMusicPerformance8::Init** method, or the **IDirectMusicPerformance8::InitAudio** method to create a DirectMusic object.

The methods of the **IDirectMusic8** interface can be organized into the following groups:

Activation	**Activate**
Buffers	**CreateMusicBuffer**
Linkage	**SetDirectSound**
Ports	**CreatePort**
	EnumPort
	GetDefaultPort
Timing	**EnumMasterClock**
	GetMasterClock
	SetExternalMasterClock
	SetMasterClock

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTMUSIC8** type is defined as a pointer to the **IDirectMusic8** interface.

```
typedef IDirectMusic8 *LPDIRECTMUSIC8;
```

Requirements

Header: Declared in dmusicc.h.

IDirectMusic8::Activate

The **IDirectMusic8::Activate** method activates or deactivates all ports created from this interface.

```
HRESULT Activate(
  BOOL fEnable
);
```

Parameters

fEnable
Switch to activate (TRUE) or deactivate (FALSE) all port objects created in this instance of DirectMusic.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DSERR_NODRIVER, indicating that no sound driver is present.

Remarks

Applications should call **IDirectMusic8::Activate**(FALSE) when they lose input focus if they do not need to play music in the background. This allows another application that has the input focus to have access to the ports. When the application has input focus again, it should call **Activate**(TRUE) to enable all its allocated ports.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort8::Activate

IDirectMusic8::CreateMusicBuffer

Creates a DirectMusicBuffer object to hold music messages being sequenced to the port. Most applications do not need to call this method directly because buffer management is handled by the performance when a port is added.

```
HRESULT CreateMusicBuffer(
  LPDMUS_BUFFERDESC pBufferDesc,
  LPDIRECTMUSICBUFFER *ppBuffer,
  LPUNKNOWN pUnkOuter
);
```

Parameters

pBufferDesc

Address of the **DMUS_BUFFERDESC** structure that contains the description of the music buffer to be created. The application must initialize the **dwSize** member of this structure before passing the pointer.

ppBuffer

Address of a variable that receives an **IDirectMusicBuffer8** interface pointer.

pUnkOuter

Address of the controlling object's **IUnknown** interface for COM aggregation. Because aggregation is not currently supported, this value must be set to NULL.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
CLASS_E_NOAGGREGATION
E_NOINTERFACE
E_OUTOFMEMORY
E_POINTER

Requirements

Header: Declared in dmusicc.h.

IDirectMusic8::CreatePort

Creates an object for a DirectMusic port.

```
HRESULT CreatePort(
  REFCLSID rclsidPort,
  LPDMUS_PORTPARAMS pPortParams,
  LPDIRECTMUSICPORT *ppPort,
  LPUNKNOWN pUnkOuter
);
```

Parameters

rclsidPort

Reference to (C++) or address of (C) the GUID that identifies the port for which the **IDirectMusicPort8** interface is to be created. The GUID is retrieved through the **IDirectMusic8::EnumPort** method. If it is GUID_NULL, the returned port is the default port. For more information, see Default Port.

pPortParams

Address of a **DMUS_PORTPARAMS8** structure that contains parameters for the port. The **dwSize** member of this structure must be initialized before the method is called.

ppPort

Address of a variable that receives an **IDirectMusicPort** interface pointer.

pUnkOuter

Address of the controlling object's **IUnknown** interface for COM aggregation. Because aggregation is not currently supported, this value must be NULL.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if a requested parameter is not available.

If it fails, the method can return one of the following error values:

DMUS_E_DSOUND_NOT_SET

E_INVALIDARG

CLASS_E_NOAGGREGATION

E_NOINTERFACE

E_OUTOFMEMORY

E_POINTER

Remarks

By default, the port is inactive when it is created. It must be activated by a call to **IDirectMusic8::Activate** or **IDirectMusicPort8::Activate**.

If not all parameters could be obtained, the **DMUS_PORTPARAMS8** structure is changed as follows to match the available parameters of the port.

On entry, the **dwValidParams** member of the structure indicates which members in the structure are valid. If the flag is not set for a member of the structure, a default value is set for that parameter when the port is created.

On return, the flags in **dwValidParams** show which port parameters were set. If a particular parameter was not requested but was set to the default, that flag is added to those passed in.

If the port supports a specified parameter but the given value for the parameter is out of range, the parameter value in **pPortParams* is changed. In this case, the flag in **dwValidParams** remains set, but S_FALSE is returned to indicate that the value has been changed.

The following code example shows how an application can request reverb capabilities and determine whether they were obtained. For an alternative way of checking and setting port properties, see Property Sets for DirectMusic Ports.

```
DMUS_PORTPARAMS8 params;
```

```
ZeroMemory(&params, sizeof(params));
params.dwSize = sizeof(params);
params.dwValidParams = DMUS_PORTPARAMS_EFFECTS;
params.dwEffectFlags = DMUS_EFFECT_REVERB;
HRESULT hr = pDirectMusic->CreatePort(guidPort, &params,
      &port, NULL);
if (SUCCEEDED(hr))
{
   fGotReverb = TRUE;
   if (hr == S_FALSE)
   {
      if (!(params.dwValidParams & DMUS_PORTPARAMS_EFFECTS))
      {
         // Device does not support any effects.
         fGotReverb = FALSE;
      }
      else if (!(params.dwEffectFlags & DMUS_EFFECT_REVERB))
      {
         // Device understands effects,
         // but could not allocate reverb.
         fGotReverb = FALSE;
      }
   }
}
```

Requirements

Header: Declared in dmusicc.h.

IDirectMusic8::EnumMasterClock

Enumerates the clocks that DirectMusic can use as the master clock. Each time it is called, this method retrieves information about a single clock.

```
HRESULT EnumMasterClock(
  DWORD dwIndex,
  LPDMUS_CLOCKINFO lpClockInfo
);
```

Parameters

dwIndex

Index of the clock for which the description is to be returned. This parameter should be 0 on the first call, and then incremented in each subsequent call until S_FALSE is returned.

lpClockInfo
Address of a **DMUS_CLOCKINFO8** structure that receives the description of the clock. The application must initialize the **dwSize** member of this structure before passing the pointer.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if there is no clock with that index number.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_NOINTERFACE
E_POINTER

Remarks

Applications should not rely on or store the index number of a clock. Rebooting or adding and removing hardware can cause the index number of a clock to change.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusic8::SetMasterClock, **IDirectMusic8::GetMasterClock**

IDirectMusic8::EnumPort

Enumerates and retrieves the capabilities of the DirectMusic ports connected to the system. Each time it is called, this method retrieves information about a single port.

```
HRESULT EnumPort(
  DWORD dwIndex,
  LPDMUS_PORTCAPS pPortCaps
);
```

Parameters

dwIndex
Index of the port for which the capabilities are to be returned. This parameter should be 0 on the first call, and then incremented in each subsequent call until S_FALSE is returned.

pPortCaps

Address of the **DMUS_PORTCAPS** structure that receives the capabilities of the port. The **dwSize** member of this structure must be initialized before the pointer is passed.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if there is no port with that index value.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_NOINTERFACE
E_POINTER

Remarks

Applications should not rely on or store the index number of a port. Rebooting or adding or removing ports could cause the index number of a port to change.

Requirements

Header: Declared in dmusicc.h.

IDirectMusic8::GetDefaultPort

Retrieves the GUID of the default output port. This is the port to be created if GUID_NULL is passed to **IDirectMusic8::CreatePort**.

```
HRESULT GetDefaultPort(
  LPGUID pguidPort
);
```

Parameters

pguidPort
Address of a variable that receives the default port GUID.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusicc.h.

See Also

Default Port

IDirectMusic8::GetMasterClock

Retrieves the GUID and a pointer to the **IReferenceClock** interface for the clock that is currently set as the DirectMusic master clock.

```
HRESULT GetMasterClock(
  LPGUID pguidClock,
  IReferenceClock **ppReferenceClock
);
```

Parameters

pguidClock
Address of a variable that receives the GUID of the master clock. The application can pass NULL if this value is not desired.

ppReferenceClock
Address of a variable that receives the **IReferenceClock** interface pointer for this clock. The application can pass NULL if this value is not desired.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_NOINTERFACE
E_POINTER

Remarks

The **IReferenceClock** interface pointer must be released after the application has finished using the interface.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusic8::SetMasterClock

IDirectMusic8::SetDirectSound

Connects DirectMusic to a DirectSound object for output from the synthesizer.

This method is not used by most applications. The DirectSound object is normally created and connected to the performance by **IDirectMusicPerformance8::InitAudio**.

```
HRESULT SetDirectSound(
  LPDIRECTSOUND pDirectSound,
  HWND hWnd
);
```

Parameters

pDirectSound

Address of the **IDirectSound8** interface to use for output. If this parameter is NULL, the method creates a DirectSound object and sets the DSSCL_PRIORITY cooperative level. (See Remarks.) If this parameter contains an **IDirectSound** pointer, the caller is responsible for setting the cooperative level.

hWnd

Window handle to the DirectSound object created by this call. If this value is NULL, the current foreground window is set as the focus window. See Remarks.

If *pDirectSound* is a valid interface, this parameter is ignored because it is the caller's responsibility to supply a valid window handle in the call to **IDirectSound8::SetCooperativeLevel**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_DSOUND_ALREADY_SET.

Remarks

The specified DirectSound object is the one used for rendering audio on all ports. This default can be overridden by using the **IDirectMusicPort8::SetDirectSound** method.

Whenever the **IDirectMusic8::SetDirectSound** method is called, any existing DirectSound object is released.

When *pDirectSound* is NULL, a new DirectSound object is not created until a port that uses DirectSound is activated, and the DirectSound object is automatically released when the last port using it is deactivated.

If you created the DirectSound object yourself, you can release it by calling this method with NULL in the *pDirectSound* parameter after deactivating all ports. (It is an error to call **SetDirectSound** on an active port.)

You can pass NULL in the *hWnd* parameter to pass the current foreground window handle to DirectSound. However, do not assume that the application window is in the foreground during initialization. In general, the top-level application window handle should be passed to DirectMusic and DirectSound. See the Remarks for **IDirectSound8::SetCooperativeLevel**.

Requirements

Header: Declared in dmusicc.h.

IDirectMusic8::SetExternalMasterClock

Sets the DirectMusic master clock to an existing clock object. There is only one master clock for all DirectMusic applications.

```
HRESULT SetExternalMasterClock(
  IReferenceClock *pClock
);
```

Parameters

pClock
IReferenceClock interface pointer that specifies the clock.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_PORTS_OPEN.

Remarks

If another running application is also using DirectMusic, it is not possible to change the master clock until that application is shut down.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusic8::GetMasterClock, **IDirectMusic8::EnumMasterClock**, **IDirectMusic8::SetMasterClock**,

IDirectMusic8::SetMasterClock

Sets the DirectMusic master clock to a clock identified by a GUID obtained through the **IDirectMusic8::EnumMasterClock** call. There is only one master clock for all DirectMusic applications.

```
HRESULT SetMasterClock(
  REFGUID rguidClock
);
```

Parameters

rguidClock

Reference to (C++) or address of (C) the GUID that identifies the clock to set as the master clock for DirectMusic. This parameter must be a GUID returned by the **IDirectMusic8::EnumMasterClock** method.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_PORTS_OPEN.

Remarks

If another running application is also using DirectMusic, it is not possible to change the master clock until that application is shut down.

Most applications do not need to call **SetMasterClock**. It should not be called unless there is a need to synchronize tightly with a hardware timer other than the system clock.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusic8::SetExternalMasterClock, **IDirectMusic8::GetMasterClock**, **IDirectMusic8::EnumMasterClock**

IDirectMusicAudioPath8

The **IDirectMusicAudioPath8** interface represents the stages of data flow from the performance to the primary sound buffer. The audiopath can be created from an audiopath configuration object by using the **IDirectMusicPerformance8::CreateAudioPath** method. A standard audiopath can

be created by using **IDirectMusicPerformance8::CreateStandardAudioPath**. A standard default path can also be created by **IDirectMusicPerformance8::InitAudio** and then retrieved by using **IDirectMusicPerformance8::GetDefaultAudioPath**.

The **IDirectMusicAudioPath8** interface can be passed to **IDirectMusicPerformance8::PlaySegmentEx** to play the segment on that audiopath.

IDirectMusicAudioPath8 is a type definition for **IDirectMusicAudioPath**. The two interface names are interchangeable.

The **IDirectMusicAudioPath8** interface has the following methods.

IDirectMusicAudioPath8	**Activate**
	ConvertPChannel
	GetObjectInPath
	SetVolume

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicAudioPath8::Activate

Activates or deactivates the audiopath.

```
HRESULT Activate(
  BOOL fActivate
);
```

Parameters

fActivate

Boolean that specifies whether to activate (TRUE) or deactivate (FALSE) the path.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the audiopath is already in the requested state.

Remarks

The behavior of this method is different from that of **IDirectMusicPort8::Activate**. When a port is deactivated, it no longer produces sound, but the performance can continue playing segments. When an audiopath is deactivated, all playback stops and any attempt to play a segment will fail.

Requirements

Header: Declared in dmusici.h.

IDirectMusicAudioPath8::ConvertPChannel

Translates between a performance channel in a segment and the equivalent channel allocated in the performance for the audiopath. This method is not typically needed by applications, but can be used by components such as tracks.

```
HRESULT ConvertPChannel(
  DWORD dwPChannelIn,
  DWORD *pdwPChannelOut
);
```

Parameters

dwPChannelIn
Value that specifies the performance channel to convert.

pdwPChannelOut
Address of a **DWORD** variable that receives the virtual performance channel.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
E_POINTER

Requirements

Header: Declared in dmusici.h.

IDirectMusicAudioPath8::GetObjectInPath

Retrieves an interface for an object in the audiopath.

```
RESULT GetObjectInPath(
  DWORD dwPChannel,
  DWORD dwStage,
  DWORD dwBuffer,
  REFGUID guidObject,
  DWORD dwIndex,
  REFGUID iidInterface,
  void ** ppObject
);
```

Parameters

dwPChannel

Performance channel to search, or DMUS_PCHANNEL_ALL to search all channels. The first channel is numbered 0. See Remarks.

dwStage

Stage in the path. Can be one of the following values.

DMUS_PATH_AUDIOPATH_GRAPH
Get the audiopath toolgraph. One is created if none exists.

DMUS_PATH_AUDIOPATH_TOOL
Get a tool from the audiopath toolgraph.

DMUS_PATH_BUFFER
Get a DirectSound buffer.

DMUS_PATH_BUFFER_DMO
Get a DMO in a buffer. For information on DMO interfaces, see **IDirectSoundBuffer8::GetObjectInPath**.

DMUS_PATH_MIXIN_BUFFER
Get a global mix-in buffer.

DMUS_PATH_MIXIN_BUFFER_DMO
Get a DMO in a global mix-in buffer.

DMUS_PATH_PERFORMANCE
Get the performance.

DMUS_PATH_PERFORMANCE_GRAPH
Get the performance toolgraph. One is created if none exists.

DMUS_PATH_PERFORMANCE_TOOL
Get a tool from the performance toolgraph.

DMUS_PATH_PORT
Get the synthesizer.

DMUS_PATH_PRIMARY_BUFFER
Get the primary buffer.

dwBuffer
If *dwStage* is DMUS_PATH_BUFFER_DMO or DMUS_PATH_MIXIN_BUFFER_DMO, the index of the buffer in which that DMO resides. If *dwStage* is DMUS_PATH_BUFFER or DMUS_PATH_MIXIN_BUFFER, the index of the buffer. Otherwise must be 0.

guidObject
Class identifier of the object, or GUID_All_Objects to search for an object of any class. This parameter is ignored if only a single class of object can exist at the stage specified by *dwStage*, and can be set to GUID_NULL.

dwIndex
Index of the object within a list of matching objects. Set to 0 to find the first matching object. If *dwStage* is DMUS_PATH_BUFFER or DMUS_PATH_MIXIN_BUFFER, this parameter is ignored, and the buffer index is specified by *dwBuffer*.

iidInterface
Identifier of the desired interface, such as IID_IDirectMusicTool.

ppObject
Address of a variable that receives a pointer to the requested interface.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND

E_INVALIDARG

E_OUTOFMEMORY

E_NOINTERFACE

E_POINTER

Remarks

The value in *dwPChannel* must be 0 for any stage that is not channel-specific. Objects in the following stages are channel-specific and can be retrieved by setting a channel number or DMUS_PCHANNEL_ALL in *dwPChannel*:

DMUS_PATH_AUDIOPATH_TOOL
DMUS_PATH_BUFFER
DMUS_PATH_BUFFER_DMO
DMUS_PATH_PERFORMANCE_TOOL
DMUS_PATH_PORT
DMUS_PATH_SEGMENT_TOOL

The precedence of the parameters in filtering out unwanted objects is as follows:

1. *dwStage*.
2. *guidObject*. If this value is not GUID_All_Objects, only objects whose class identifier equals *guidObject* are searched. However, this parameter is ignored for stages where only a single class of object can exist, such as DMUS_PATH_AUDIOPATH_GRAPH.
3. *dwPChannel*. If the stage is channel-specific and this value is not DMUS_PCHANNEL_ALL, only objects on the channel are searched.
4. *dwBuffer*. This is used only if *dwStage* is DMUS_PATH_BUFFER, DMUS_PATH_MIXIN_BUFFER, DMUS_PATH_BUFFER_DMO, or DMUS_PATH_MIXIN_BUFFER_DMO.
5. *dwIndex*.

If a matching object is found but the interface specified by *iidInterface* cannot be obtained, the method fails.

The following sample shows how to enumerate the buffers in an audiopath. The **_stprintf** function is declared in Tchar.h.

```
void DumpAudioPathBuffers(
        IDirectMusicAudioPath *pDirectMusicAudioPath)
{
    TCHAR tcstrText[256];
    DWORD dwBuffer = 0;
    IDirectSoundBuffer *pDirectSoundBuffer;

    while(S_OK == pDirectMusicAudioPath->GetObjectInPath(
        DMUS_PCHANNEL_ALL, DMUS_PATH_BUFFER, dwBuffer,
        GUID_NULL, 0, IID_IDirectSoundBuffer,
        (void**) &pDirectSoundBuffer))
    {
        _stprintf( tcstrText, _T("Found buffer %x.\n"),
                pDirectSoundBuffer);
        OutputDebugString( tcstrText );
        dwBuffer++;
        pDirectSoundBuffer->Release();
    }

    if( dwBuffer == 0 )
    {
        OutputDebugString( _T("No buffers in audiopath.\n") );
    }
}
```

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegmentState8::GetObjectInPath, Retrieving Objects from an Audiopath

IDirectMusicAudioPath8::SetVolume

Sets the audio volume on the path.

```
HRESULT SetVolume(
  long lVolume,
  DWORD dwDuration
);
```

Parameters

lVolume
Value that specifies the attenuation, in hundredths of a decibel. This value must be in the range from -9600 to 0. Zero is full volume.

dwDuration
Value that specifies the time, in milliseconds, over which the volume change takes place. A value of 0 ensures maximum efficiency.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_INVALIDARG

Requirements

Header: Declared in dmusici.h.

IDirectMusicBand8

The **IDirectMusicBand8** interface represents a DirectMusic band object. A band is used to set the instrument choices and mixer settings for a set of performance channels. For an overview, see Using Bands. Bands can be stored directly in their own files or embedded in a style's band list or a segment's band track.

IDirectMusicBand8 is a define for **IDirectMusicBand**. The two interface names are interchangeable.

The DirectMusicBand object also supports the **IPersistStream** and **IDirectMusicObject8** interfaces for loading its data.

The **IDirectMusicBand8** interface has the following methods:

Segment creation	**CreateSegment**
Instrument data	**Download**
	Unload

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicBand8::CreateSegment

Creates a DirectMusicSegment object that can be played in order to perform the volume, pan, transposition, and patch change commands in the band dynamically.

```
HRESULT CreateSegment(
  IDirectMusicSegment** ppSegment
);
```

Parameters

ppSegment
Address of a variable that receives a pointer to the created segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_OUTOFMEMORY
E_POINTER

Remarks

For an example of creating a segment from a band, see Making Band Changes Programmatically.

Requirements

Header: Declared in dmusici.h.

IDirectMusicBand8::Download

Downloads the DLS data for instruments in the band to a performance object.

```
HRESULT Download(
  IDirectMusicPerformance* pPerformance
);
```

Parameters

pPerformance
Performance to which instruments are to be downloaded. The performance manages the mapping of performance channels to DirectMusic ports.

Return Values

If the method succeeds, the return value is S_OK, or DMUS_S_PARTIALDOWNLOAD. (See Remarks.)

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_OUTOFMEMORY
E_POINTER

Remarks

The method downloads each instrument in the band by calling the **IDirectMusicPerformance8::DownloadInstrument** method. **DownloadInstrument**, in turn, uses the performance channel of the instrument to find the appropriate port, and then calls the **IDirectMusicPort8::DownloadInstrument** method on that port.

After a band has been downloaded, the instruments in the band can be selected, either individually with program-change MIDI messages, or all at once by playing a band segment created through a call to the **IDirectMusicBand8::CreateSegment** method.

Because a downloaded band uses synthesizer resources, it should be unloaded when no longer needed by using the **IDirectMusicBand8::Unload** method.

This method may return S_OK even though the port does not support DLS.

If the download completely fails, DMUS_E_NOT_INIT is returned. This usually means that the performance was not properly connected to an initialized port. Because this is a complete failure, there is no need to call **IDirectMusicBand8::Unload** later.

If the download partially succeeds, DMUS_S_PARTIALDOWNLOAD is returned. This means that some of the instruments successfully downloaded and others did not. This usually occurs because of programming error setting up the performance and port. The best way to find the problem is to set debug traces to 1 for Dmime.dll, Dmband.dll, and Dmsynth.dll. See Debugging DirectX Audio Projects.

The following are some common causes of a partial download:

- The band has instruments on performance channels that have not been set up on the performance (by using **IDirectMusicPerformance8::AssignPChannelBlock**).
- The band has instruments on performance channels that are on channel groups not allocated on the port.
- The band has instruments in a DLS format incompatible with the synthesizer they are being downloaded to.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicBand8::Unload

IDirectMusicBand8::Unload

Unloads the DLS data for instruments in the band previously downloaded by **IDirectMusicBand8::Download**.

```
HRESULT Unload(
  IDirectMusicPerformance* pPerformance
);
```

Parameters

pPerformance
Performance from which to unload instruments.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
DMUS_E_NOT_FOUND

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPort8::UnloadInstrument

IDirectMusicBuffer8

The **IDirectMusicBuffer8** interface represents a buffer containing time stamped data (typically in the form of MIDI messages) to be sequenced by a port. Unlike a segment, the buffer contains a small amount of data (typically less than 200 milliseconds) over which the application has control at event granularity.

IDirectMusicBuffer8 is a type definition for **IDirectMusicBuffer**. The two interface names are interchangeable.

Unless your application is doing its own sequencing, you do not need to use the methods of this interface.

Buffer objects are completely independent of port objects until the buffer is passed to the port by a call to the **IDirectMusicPort8::PlayBuffer** or the **IDirectMusicPort8::Read** method. The application is then free to reuse the buffer.

The methods of the **IDirectMusicBuffer8** interface can be organized in the following groups:

Data	**Flush**
	GetNextEvent
	GetRawBufferPtr
	PackStructured
	PackUnstructured
	ResetReadPtr
Parameters	**GetBufferFormat**
	GetMaxBytes
	GetUsedBytes
	SetUsedBytes
Time	**GetStartTime**
	SetStartTime
	TotalTime

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTMUSICBUFFER8** type is defined as a pointer to the **IDirectMusicBuffer8** interface.

```
typedef IDirectMusicBuffer8 *LPDIRECTMUSICBUFFER8;
```

Requirements

Header: Declared in dmusicc.h.

IDirectMusicBuffer8::Flush

Discards all data in the buffer.

```
HRESULT Flush();
```

Parameters

None.

Return Values

The method always returns S_OK.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicBuffer8::GetBufferFormat

Retrieves the GUID that represents the buffer format.

```
HRESULT GetBufferFormat(
  LPGUID pGuidFormat
);
```

Parameters

pGuidFormat
Address of a variable that receives the GUID of the buffer format.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

If the format was not specified when the buffer was created, KSDATAFORMAT_SUBTYPE_DIRECTMUSIC is returned in **pGuidFormat*.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusic8::CreateMusicBuffer, **DMUS_EVENTHEADER**

IDirectMusicBuffer8::GetMaxBytes

Retrieves the number of bytes that can be stored in the buffer.

```
HRESULT GetMaxBytes(
  LPDWORD pcb
);
```

Parameters

pcb

Address of a variable to contain the maximum number of bytes that the buffer can hold.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicBuffer8::GetNextEvent

Returns information about the next message in the buffer and advances the read pointer.

```
HRESULT GetNextEvent(
  LPREFERENCE_TIME prt,
  LPDWORD pdwChannelGroup,
  LPDWORD pdwLength,
  LPBYTE* ppData
);
```

Parameters

prt
Address of a variable that receives the time of the message.

pdwChannelGroup
Address of a variable that receives the channel group of the message.

pdwLength
Address of a variable that receives the length, in bytes, of the message.

ppData
Address of a variable that receives a pointer to the message data.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if there are no messages in the buffer.

If it fails, the method can return E_POINTER.

Remarks

Any of the passed pointers can be NULL if the item is not needed.

The pointer returned in *ppData* is valid only for the lifetime of the buffer object.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8::ResetReadPtr

IDirectMusicBuffer8::GetRawBufferPtr

Returns a pointer to the underlying buffer data structure.

```
HRESULT GetRawBufferPtr(
  LPBYTE* ppData
);
```

Parameters

ppData
Address of a variable that receives a pointer to the buffer's data.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

This method returns a pointer to the raw data of the buffer. The format of the data depends on the implementation. The lifetime of the data is the same as the lifetime of the buffer object; therefore, the returned pointer should not be held after the next call to the **IDirectMusicBuffer8::Release** method.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicBuffer8::GetStartTime

Retrieves the start time of the data in the buffer, relative to the master clock.

```
HRESULT GetStartTime(
  LPREFERENCE_TIME prt
);
```

Parameters

prt
Address of a variable that receives the start time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_BUFFER_EMPTY
E_POINTER

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8::SetStartTime, **IDirectMusicBuffer8::TotalTime**

IDirectMusicBuffer8::GetUsedBytes

Retrieves the number of bytes of data in the buffer.

```
HRESULT GetUsedBytes(
  LPDWORD pcb
);
```

Parameters

pcb
Address of a variable that receives the number of used bytes.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8::SetUsedBytes

IDirectMusicBuffer8::PackStructured

Inserts fixed-length data (typically a MIDI channel message), along with timing and routing information, into the buffer.

```
HRESULT PackStructured(
  REFERENCE_TIME rt,
  DWORD dwChannelGroup,
  DWORD dwChannelMessage
);
```

Parameters

rt
Absolute time of the message. See Remarks.

dwChannelGroup

Channel group to which the data belongs.

dwChannelMessage
Data (MIDI message) to pack.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_INVALID_EVENT
E_OUTOFMEMORY

Remarks

At least 32 bytes (the size of **DMUS_EVENTHEADER** plus *dwChannelMessage*) must be free in the buffer.

The *rt* parameter must contain the absolute time at which the data is to be sent to the port. To play a message immediately, retrieve the time from the latency clock, and use this as *rt*. See **IDirectMusicPort8::GetLatencyClock**.

Messages stamped with the same time do not necessarily play in the same order in which they were placed in the buffer.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8::PackUnstructured

IDirectMusicBuffer8::PackUnstructured

Inserts unstructured data (typically a MIDI system exclusive message), along with timing and routing information, into the buffer.

```
HRESULT PackUnstructured(
  REFERENCE_TIME rt,
  DWORD dwChannelGroup,
  DWORD cb,
  LPBYTE lpb
);
```

Parameters

rt

Absolute time of the message.

dwChannelGroup
Channel group to which the message belongs.

cb
Size of the data, in bytes.

lpb
Address of a buffer containing the data.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY
E_POINTER

Remarks

This method can be used to send any kind of data to the port.

At least 28 bytes (the size of **DMUS_EVENTHEADER**) plus the size of the data, padded to a multiple of 4 bytes, must be free in the buffer. The buffer space required can be obtained by using the **DMUS_EVENT_SIZE**(*cb*) macro, where *cb* is the size of the data.

The *rt* parameter must contain the absolute time at which the data is to be sent to the port. To play a message immediately, retrieve the time from the latency clock, and use this as *rt*. See **IDirectMusicPort8::GetLatencyClock**.

Messages stamped with the same time do not necessarily play in the same order in which they were placed in the buffer.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8::PackStructured

IDirectMusicBuffer8::ResetReadPtr

Sets the read pointer to the start of the data in the buffer.

HRESULT ResetReadPtr()

Parameters

None.

Return Values

The method always returns S_OK.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8::GetNextEvent

IDirectMusicBuffer8::SetStartTime

Sets the start time of the data in the buffer, relative to the master clock.

```
HRESULT SetStartTime(
  REFERENCE_TIME rt
);
```

Parameters

rt
New start time for the buffer.

Return Values

The method always returns S_OK.

Remarks

Events already in the buffer are time stamped relative to the start time and play at the same offset from the new start time.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8::GetStartTime

IDirectMusicBuffer8::SetUsedBytes

Sets the number of bytes of data in the buffer.

```
HRESULT SetUsedBytes(
  DWORD cb
);
```

Parameters

cb

Number of valid data bytes in the buffer.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_BUFFER_FULL.

Remarks

This method allows an application to repack a buffer manually. Normally, this should be done only if the data format in the buffer is different from the default format provided by DirectMusic.

The method fails if the specified number of bytes exceeds the maximum buffer size, as returned by the **IDirectMusicBuffer8::GetMaxBytes** method.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8::GetUsedBytes

IDirectMusicBuffer8::TotalTime

Returns the total time spanned by the data in the buffer.

```
HRESULT TotalTime(
  LPREFERENCE_TIME prtTime
);
```

Parameters

prtTime

Address of a variable that receives the total time spanned by the buffer, in units of 100 nanoseconds.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8::GetStartTime

IDirectMusicChordMap8

The **IDirectMusicChordMap8** interface represents a chordmap. Chordmaps provide the composer (represented by the **IDirectMusicComposer8** interface) with the information needed to create chord progressions for segments to be composed. Chordmaps can also be used to change the chords in an existing segment.

The DirectMusicChordMap object also supports the **IDirectMusicObject8** and **IPersistStream** interfaces for loading its data.

IDirectMusicChordMap8 is a type definition for **IDirectMusicChordMap**. The two interface names are interchangeable.

The interface has the following method:

IDirectMusicChordMap8	**GetScale**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicChordMap8::GetScale

Retrieves the scale associated with the chordmap.

```
HRESULT GetScale(
  DWORD* pdwScale
);
```

Parameters

pdwScale
Address of a variable that receives the scale value.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be E_POINTER.

Remarks

The scale is defined by the bits in a **DWORD**, split into a scale pattern (lower 24 bits) and a root (upper 8 bits). For the scale pattern, the low bit (0x0001) is the lowest note in the scale, the next higher (0x0002) is a semitone higher, and so on for two octaves. The upper 8 bits give the root of the scale as an integer between 0 and 23 (low C to middle B).

Requirements

Header: Declared in dmusici.h.

IDirectMusicCollection8

The **IDirectMusicCollection8** interface manages an instance of a DLS file. The collection provides methods to access instruments and download them to the synthesizer by means of the **IDirectMusicPort8** interface.

IDirectMusicCollection8 is a type definition for **IDirectMusicCollection**. The two interface names are interchangeable.

The DirectMusicCollection object also supports the **IDirectMusicObject8** and **IPersistStream** interfaces for loading its data.

For more information on how to work with collections, see Using Instrument Collections.

The **IDirectMusicCollection8** interface has the following methods:

Instruments	**EnumInstrument**
	GetInstrument

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusicc.h.

IDirectMusicCollection8::EnumInstrument

Retrieves the patch and name of an instrument by its index in the collection.

```
HRESULT EnumInstrument(
  DWORD dwIndex,
  DWORD* pdwPatch,
  LPWSTR pwszName,
  DWORD dwNameLen
);
```

Parameters

dwIndex
Index of the instrument in the collection.

pdwPatch
Address of a variable that receives the patch number.

pwszName
Address of a buffer that receives the instrument name. Can be NULL if the name is not wanted.

dwNameLen
Number of **WCHAR** values in the instrument name buffer.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if there is no instrument with that index number.

If it fails, the method can return one of the following error values:

E_FAIL
E_OUTOFMEMORY
E_POINTER

Remarks

To enumerate all instruments in a collection, start with a *dwIndex* of 0 and increment until **EnumInstrument** returns S_FALSE.

The patch number returned in *pdwPatch* describes the full patch address, including the MIDI parameters for MSB and LSB bank select. For more information, see MIDI Channel Messages.

Although the ordering of the enumeration is consistent within one instance of a DLS collection, it has no relationship to the ordering of instruments in the file, their patch numbers, or their names.

For an example of instrument enumeration, see Working with Instruments.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicCollection8::GetInstrument

Retrieves an instrument from a collection by its patch number.

```
HRESULT GetInstrument(
  DWORD dwPatch,
  IDirectMusicInstrument** ppInstrument
);
```

Parameters

dwPatch
Instrument patch number.

ppInstrument
Address of a variable that receives a pointer to the **IDirectMusicInstrument8** interface.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_INVALIDPATCH
E_FAIL
E_OUTOFMEMORY
E_POINTER

Remarks

The patch number passed in *dwPatch* describes the full patch address, including the MIDI parameters for MSB and LSB bank select. MSB is shifted left 16 bits, and LSB is shifted left 8 bits. For more information, see MIDI Channel Messages.

In addition, the high bit is set (0x80000000) if the instrument is specifically a drum kit intended to be played on MIDI channel 10.

For an example of how this method is used, see Working with Instruments.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicComposer8

The **IDirectMusicComposer8** interface permits access to the composition engine. In addition to building new segments from templates and chordmaps, the composer can generate transitions between different segments. It can also apply a chordmap to an existing segment, thus altering the chord progression and the mood of the music.

IDirectMusicComposer8 is a define for **IDirectMusicComposer.** The two interface names are interchangeable.

The methods of the **IDirectMusicComposer8** interface can be grouped as follows:

Changing chordmaps	**ChangeChordMap**
Composing ordinary segments	**ComposeSegmentFromShape**
	ComposeSegmentFromTemplate
Composing template segments	**ComposeTemplateFromShape**
Composing transition segments	**AutoTransition**
	ComposeTransition

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicComposer8::AutoTransition

Composes a transition from inside a performance's primary segment (or from silence) to another segment, and then cues the transition and the second segment to play.

```
HRESULT AutoTransition(
  IDirectMusicPerformance* pPerformance,
  IDirectMusicSegment* pToSeg,
  WORD wCommand,
  DWORD dwFlags,
  IDirectMusicChordMap* pChordMap,
  IDirectMusicSegment** ppTransSeg,
  IDirectMusicSegmentState** ppToSegState,
  IDirectMusicSegmentState** ppTransSegState
);
```

Parameters

pPerformance
Performance in which to make the transition.

pToSeg
Segment to which the transition should smoothly flow. See Remarks.

wCommand
Embellishment to use when composing the transition. This can be one of the **DMUS_COMMANDT_TYPES** enumeration, or another value defined by the content provider. If this value is DMUS_COMMANDT_ENDANDINTRO, the method composes a segment containing both an ending to the primary segment and an introduction to *pToSeg*.

dwFlags
Composition options. See **DMUS_COMPOSEF_FLAGS**.

pChordMap
Pointer to the **IDirectMusicChordMap8** interface of the chordmap to be used when composing the transition.

ppTransSeg
Address of a variable that receives a pointer to the created segment. This value can be NULL, in which case the pointer is not returned.

ppToSegState
Address of a variable that receives a pointer to the segment state created by the performance (*pPerformance*) for the segment following the transition (*pToSeg*). See Remarks.

ppTransSegState
Address of a variable that receives a pointer to the segment state created by the performance (*pPerformance*) for the created segment (*ppTransSeg*). See Remarks.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NO_MASTER_CLOCK
E_INVALIDARG
E_POINTER

Remarks

The value in *pToSeg* can be NULL as long as *dwFlags* does not include DMUS_COMPOSEF_MODULATE. If *pToSeg* is NULL or does not contain a style track (as would be the case if it is based on a MIDI file), introductory embellishments are not valid. If the currently playing segment is NULL or does not contain a style track, then fill, break, end, and groove embellishments are not valid. If no style track is available either in the currently playing segment or in the one represented by *pToSeg*, all embellishments are invalid, and no transition occurs. In that case, both *ppTransSeg* and *ppTransSegState* return NULL, but the method succeeds and cues the segment represented by *pToSeg*, if that pointer is not NULL.

The value in *pChordMap* can be NULL. If it is, an attempt is made to obtain a chordmap from a chordmap track, first from *pToSeg*, and then from the performance's primary segment. If neither of these segments contains a chordmap track, the chord occurring at the current time in the primary segment is used as the chord in the transition.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicComposer8::ComposeTransition, Using Transitions

IDirectMusicComposer8::ChangeChordMap

Modifies the chords and scale pattern of an existing segment to reflect a new chordmap.

```
HRESULT ChangeChordMap(
  IDirectMusicSegment* pSegment,
  BOOL fTrackScale,
  IDirectMusicChordMap* pChordMap
);
```

Parameters

pSegment
Pointer to the **IDirectMusicSegment8** interface of the segment in which to change the chordmap. This segment must contain a chordmap track and a style.

fTrackScale
If TRUE, the method transposes all the chords to be relative to the root of the new chordmap's scale, rather than leaving their roots as they were.

pChordMap
Pointer to the **IDirectMusicChordMap8** interface of the new chordmap for the segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

The method can be called while the segment is playing.

Requirements

Header: Declared in dmusici.h.

IDirectMusicComposer8::ComposeSegmentFromShape

Creates an original segment from a style and a chordmap, based on a predefined shape. The shape represents the way chords and embellishments occur over time across the segment.

```
HRESULT ComposeSegmentFromShape(
  IDirectMusicStyle* pStyle,
  WORD wNumMeasures,
  WORD wShape,
  WORD wActivity,
  BOOL fIntro,
  BOOL fEnd,
  IDirectMusicChordMap* pChordMap,
  IDirectMusicSegment** ppSegment
);
```

Parameters

pStyle
Style from which to compose the segment.

wNumMeasures
Length, in measures, of the segment to be composed.

wShape
Shape of the segment to be composed. Possible values are of the **DMUS_SHAPET_TYPES** enumerated type.

wActivity
Rate of harmonic motion. Valid values are from 0 through 3. Lower values mean more chord changes.

fIntro
TRUE if an introduction is to be composed for the segment.

fEnd
TRUE if an ending is to be composed for the segment.

pChordMap
Pointer to the **IDirectMusicChordMap8** interface of the chordmap from which to create the segment.

ppSegment
Address of a variable that receives a pointer to the created segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicComposer8::ComposeSegmentFromTemplate,
IDirectMusicComposer8::ComposeTemplateFromShape

IDirectMusicComposer8::ComposeSegmentFromTemplate

Creates an original segment from a style, a chordmap, and a template.

HRESULT ComposeSegmentFromTemplate(
IDirectMusicStyle* *pStyle*,
IDirectMusicSegment* *pTemplate*,
WORD *wActivity*,

IDirectMusicChordMap* *pChordMap*,
IDirectMusicSegment** *ppSegment*
);

Parameters

pStyle
IDirectMusicStyle8 interface pointer that specifies the style from which to create the segment.

pTemplate
IDirectMusicSegment8 interface pointer that specifies the template from which to create the segment.

wActivity
Rate of harmonic motion. Valid values are 0 through 3. Lower values mean more chord changes.

pChordMap
IDirectMusicChordMap8 interface pointer that specifies the chordmap from which to create the segment.

ppSegment
Address of a variable that receives a pointer to the created segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_POINTER

Remarks

If *pStyle* is non-NULL, it is used in composing the segment; if it is NULL, a style is retrieved from the template specified in *pTempSeg*. Similarly, if *pChordMap* is non-NULL, it is used in composing the segment; if it is NULL, a chordmap is retrieved from the template.

If *pStyle* is NULL and there is no style track in the template, or *pChordMap* is NULL and there is no chordmap track, the method returns E_INVALIDARG.

The length of the segment is equal to the length of the template passed in.

The default start point and loop points of the created segment are 0, regardless of the values in the template segment.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicComposer8::ComposeSegmentFromShape, **IDirectMusicComposer8::ComposeTemplateFromShape**, Using Templates

IDirectMusicComposer8::ComposeTemplateFromShape

Creates a new template segment, based on a predefined shape.

```
HRESULT ComposeTemplateFromShape(
  WORD wNumMeasures,
  WORD wShape,
  BOOL fIntro,
  BOOL fEnd,
  WORD wEndLength,
  IDirectMusicSegment** ppTemplate
);
```

Parameters

wNumMeasures
Length, in measures, of the segment to be composed. This value must be greater than 0.

wShape
Shape of the segment to be composed. Possible values are of the **DMUS_SHAPET_TYPES** enumerated type.

fIntro
TRUE if an introduction is to be composed for the segment.

fEnd
TRUE if an ending is to be composed for the segment.

wEndLength
Length in measures of the ending, if one is to be composed. If *fEnd* is TRUE, this value must be greater than 0 and equal to or less than the number of measures available (that is, not used in the introduction). See also Remarks.

ppTemplate
Address of a variable that receives a pointer to the created template segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_OUTOFMEMORY
E_POINTER

Remarks

The value of *wEndLength* should not be greater than the length of the longest ending available in any style likely to be associated with this template through the **IDirectMusicComposer8::ComposeSegmentFromTemplate** method. The ending starts playing at *wEndLength* measures before the end of the segment. If the ending is less than *wEndLength* measures long, the music then reverts to the basic groove level.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicComposer8::ComposeSegmentFromTemplate, Using Templates

IDirectMusicComposer8::ComposeTransition

Composes a transition from a measure inside one segment to another.

```
HRESULT ComposeTransition(
  IDirectMusicSegment* pFromSeg,
  IDirectMusicSegment* pToSeg,
  MUSIC_TIME mtTime,
  WORD wCommand,
  DWORD dwFlags,
  IDirectMusicChordMap* pChordMap,
  IDirectMusicSegment** ppTransSeg
);
```

Parameters

pFromSeg
Segment from which to compose the transition.

pToSeg
Segment to which the transition should smoothly flow. Can be NULL if *dwFlags* does not include DMUS_COMPOSEF_MODULATE.

mtTime

Time in *pFromSeg* from which to compose the transition.

wCommand
Embellishment to use when composing the transition. This can be one of the **DMUS_COMMANDT_TYPES** enumeration, or another value defined by the content provider. If this value is DMUS_COMMANDT_ENDANDINTRO, the method composes a segment containing both an ending to *pFromSeg* and an introduction to *pToSeg*.

dwFlags
Composition options. This parameter can contain one or more of the **DMUS_COMPOSEF_FLAGS** enumerated type values.

pChordMap
Pointer to the **IDirectMusicChordMap8** interface chordmap to be used when composing the transition. See Remarks.

ppTransSeg
Address of a variable that receives a pointer to the created segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG

E_OUTOFMEMORY

E_POINTER

Remarks

The value in *pChordMap* can be NULL. If it is, an attempt is made to obtain a chordmap from a chordmap track, first from *pToSeg*, and then from *pFromSeg*. If neither of these segments contains a chordmap track, the chord occurring at *mtTime* in *pFromSeg* is used as the chord in the transition.

The composer looks for a tempo, first in *pFromSeg*, and then in *pToSeg*. If neither of those segments contains a tempo track, the tempo for the transition segment is taken from the style.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicComposer8::AutoTransition, Using Transitions

IDirectMusicContainer8

The **IDirectMusicContainer8** interface provides access to objects in a container, which is a collection of objects used by a segment or performance. The interface can be obtained when a container is loaded by a call to **IDirectMusicLoader8::GetObject** or **IDirectMusicLoader8::LoadObjectFromFile**.

When a container object is loaded, it makes all its objects available to the loader. When the container is released, all objects it refers to are released from the loader. However, any objects still in use when the container is released are not freed until explicitly released. If they are keeping a stream open, as DLS collections and streaming waves do, the stream also stays open. As a result, the container file stays locked, just as an individual wave or DLS file would.

A container can be embedded in a segment. The container is placed in the file before the segment's tracks, so it can be read and its objects installed in the loader before the tracks are loaded. When the tracks are loaded, the loader is able to supply links to referenced objects in the container.

IDirectMusicContainer8 is a type definition for **IDirectMusicContainer**. The two interface names are interchangeable.

The **IDirectMusicContainer8** interface has the following method.

Enumeration	**EnumObject**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

See Also

Containers

IDirectMusicContainer8::EnumObject

Retrieves information about an object in the container.

HRESULT EnumObject(
  REFGUID *rguidClass*,

DWORD *dwIndex*,
LPDMUS_OBJECTDESC *pDesc*,
WCHAR* *pwszAlias*
);

Parameters

rguidClass
Reference to (C++) or address of (C) the unique identifier of the object class, or GUID_DirectMusicAllTypes to obtain an object of any type. For a list of standard loadable classes, see **IDirectMusicLoader8**.

dwIndex
Index of the object among objects of class *rguidClass* in the container.

pDesc
Pointer to a **DMUS_OBJECTDESC** structure that receives a description of the object. This parameter can be NULL if no description is wanted. See Remarks.

pwszAlias
Address of a string buffer of size MAX_PATH that receives the object's alias, if it has one. An alias is a special name used by a script to refer to the object. This parameter can be NULL if no alias is wanted.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The object was enumerated.
S_FALSE	There is no object with an index of *dwIndex*.
DMUS_S_STRING_TRUNCATED	The alias is longer than MAX_PATH
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If the method fails, it can return E_POINTER.

Remarks

You must initialize the **dwSize** member of the **DMUS_OBJECTDESC** structure before passing it to the method. Other members are ignored. You cannot reduce the scope of the enumeration by, for example, specifying a value in the **wszName** member. The description returned by the method can be used to retrieve the object by calling **IDirectMusicLoader8::GetObject**. For sample code, see Containers.

Requirements

Header: Declared in dmusici.h.

IDirectMusicDownload8

The **IDirectMusicDownload8** interface represents a contiguous memory chunk used for downloading to a DLS synthesizer port.

IDirectMusicDownload8 is a type definition for **IDirectMusicDownload**. The two interface names are interchangeable.

The **IDirectMusicDownload8** interface and its contained memory chunk are created by the **IDirectMusicPortDownload8::AllocateBuffer** method. The memory can then be accessed by using the single method of this interface.

This interface is used only by applications that need to access DLS buffers directly rather than letting the performance, band, and segment objects download instrument data. For an overview, see Low-Level DLS.

The interface has the following method:

Memory access	**GetBuffer**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusicc.h.

IDirectMusicDownload8::GetBuffer

Retrieves a pointer to a buffer containing data to be downloaded.

```
HRESULT GetBuffer(
  void** ppvBuffer,
  DWORD* pdwSize
);
```

Parameters

ppvBuffer
Address of a variable that receives a pointer to the data buffer.

pdwSize
Address of a variable that receives the size of the returned buffer, in bytes.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_BUFFERNOTAVAILABLE
E_POINTER

Remarks

The method returns DMUS_E_BUFFERNOTAVAILABLE if the buffer has already been downloaded.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicDownloadedInstrument8

The **IDirectMusicDownloadedInstrument8** interface is used to identify an instrument that has been downloaded to the synthesizer by using the **IDirectMusicPort8::DownloadInstrument** or the **IDirectMusicPerformance8::DownloadInstrument** method. The interface pointer is then used to unload the instrument through a call to **IDirectMusicPort8::UnloadInstrument**. After the instrument has been unloaded, the interface pointer must be released by the application. For an example, see Working with Instruments.

IDirectMusicDownloadedInstrument8 is a type definition for **IDirectMusicDownloadedInstrument**. The two interface names are interchangeable.

The **IDirectMusicDownloadedInstrument8** interface has no methods of its own. Like all COM interfaces, it inherits the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusicc.h.

IDirectMusicGetLoader8

The **IDirectMusicGetLoader8** interface is used by an object parsing a stream when the object needs to load another object referenced by the stream. If a stream supports the loader, it must provide an **IDirectMusicGetLoader8** interface.

For an example of how to obtain the **IDirectMusicGetLoader8** interface from the stream, see **IDirectMusicGetLoader8::GetLoader**.

IDirectMusicGetLoader8 is a type definition for **IDirectMusicGetLoader**. The two interface names are interchangeable.

The **IDirectMusicGetLoader8** interface has the following method:

IDirectMusicGetLoader8	**GetLoader**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8, Loading Audio Data, Custom Loading

IDirectMusicGetLoader8::GetLoader

Retrieves a pointer to the loader object that created the stream.

```
HRESULT GetLoader8(
  IDirectMusicLoader ** ppLoader
);
```

Parameters

ppLoader

Address of a variable that receives the **IDirectMusicLoader** interface pointer. Use **QueryInterface** to obtain **IDirectMusicLoader8**. The reference count of the interface is incremented.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_NOINTERFACE.

Remarks

The following code example is from a file parser that finds a reference to an object that needs to be accessed by the loader.

```
HRESULT myGetReferencedObject(
    DMUS_OBJECTDESC *pDesc,          // Descriptor already prepared
    IStream *pIStream,               // Stream being parsed
    IDirectMusicObject **ppIObject)  // Object to be accessed
{
    IDirectMusicGetLoader *pIGetLoader;
    IDirectMusicLoader *pILoader;
    ppIObject = NULL;
    HRESULT hr = pIStream->QueryInterface(
          IID_IDirectMusicGetLoader,
          (void **) &pIGetLoader );
    if (SUCCEEDED(hr))
    {
       hr = pIGetLoader->GetLoader(&pILoader);
       if (SUCCEEDED(hr))
       {
          hr = pILoader->GetObject(pDesc, IID_DirectMusicLoader,
               (void**) ppIObject);
          pILoader->Release();
       }
       pIGetLoader->Release();
    }
    return hr;
}
```

Requirements

Header: Declared in dmusici.h.

See Also

Custom Loading

IDirectMusicGraph8

The **IDirectMusicGraph8** interface manages the loading and message flow of tools.

Graphs can occur in two places: performances and segments. The graph of tools in a performance is global in nature; it processes messages from all segments. A graph in a segment exists only for playback of that segment.

IDirectMusicGraph8 is a type definition for **IDirectMusicGraph**. The two interface names are interchangeable.

The **IDirectMusicGraph8** interface has the following methods:

Routing	**StampPMsg**
Tools	**GetTool**
	InsertTool
	RemoveTool

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicGraph8::GetTool

Retrieves a tool by index.

```
HRESULT GetTool(
  DWORD dwIndex,
  IDirectMusicTool** ppTool
);
```

Parameters

dwIndex
Zero-based index of the requested tool in the graph.

ppTool
Address of a variable that receives a pointer to the tool.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following values:

DMUS_E_NOT_FOUND

E_POINTER

Remarks

The application is responsible for releasing the retrieved tool.

Requirements

Header: Declared in dmusici.h.

IDirectMusicGraph8::InsertTool

Inserts a tool in the graph.

```
HRESULT InsertTool(
  IDirectMusicTool * pTool,
  DWORD * pdwPChannels,
  DWORD cPChannels,
  LONG lIndex
);
```

Parameters

pTool
Tool to insert.

pdwPChannels
Address of an array of PChannels on which the tool accepts messages. If the tool accepts messages on all channels, pass NULL.

cPChannels
Count of how many channels are pointed to by *pdwPChannels*. Ignored if *pdwPChannels* is NULL.

lIndex
Position at which to place the tool. This is a zero-based index from the start of the current tool list or, if it is negative, from the end of the list. If *lIndex* is out of range, the tool is placed at the beginning or end of the list. To place a tool at the end of the list, use a number for *lIndex* that is larger than the number of tools in the current tool list.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_ALREADY_EXISTS

E_OUTOFMEMORY
E_POINTER

Remarks

The reference count of the tool is incremented.

This method calls **IDirectMusicTool8::Init**.

Requirements

Header: Declared in dmusici.h.

IDirectMusicGraph8::RemoveTool

Removes a tool from the graph.

```
HRESULT RemoveTool(
  IDirectMusicTool * pTool
);
```

Parameters

pTool
Tool to be removed.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
E_POINTER

Remarks

The graph's reference to the tool object is released.

Requirements

Header: Declared in dmusici.h.

IDirectMusicGraph8::StampPMsg

Stamps a message with a pointer to the next tool that is to receive it. After processing a message, a tool must call this method.

```
HRESULT StampPMsg(
  DMUS_PMSG* pPMSG
);
```

Parameters

pPMSG

Address of a structure that contains the message to stamp. This structure is of a type derived from **DMUS_PMSG**.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_LAST_TOOL. See Remarks.

If it fails, the method can return E_POINTER.

Remarks

On entry, the **pTool** member of the **DMUS_PMSG** part of the message structure points to the current tool. **StampPMsg** uses this member to determine the current tool to find the next tool in the graph. A value of NULL represents the first tool in the graph.

The object pointed to by the **pGraph** member represents the graph that contains the tool. This is stamped inside **StampPMsg**, along with the tool itself, and can change while the message travels from the segment state to the performance because there can be multiple toolgraphs.

The value of **dwType** equals the media type of the message, and is also used to find the next tool. The media types supported are those returned by the **IDirectMusicTool8::GetMediaTypes** method.

This method calls **Release** on the current **IDirectMusicTool8** pointed to by **pTool**, replaces it with the next tool in the graph and calls **AddRef** on the new tool. It also flags the message with the correct delivery type, according to what type the next tool returns in its **IDirectMusicTool8::GetMsgDeliveryType** method. This flag determines when the message is delivered to the next tool.

Tools should not call **StampPMsg** until all other tasks have been performed. When audiopaths are in use, **StampPMsg** can have the effect of changing the value in the **dwPChannel** member of the message structure. A tool that uses this value cannot rely on it if **StampPMsg** has already been called

The implementations of this method in the DirectMusicSegmentState and DirectMusicPerformance objects always return S_OK on success. The implementation in DirectMusicGraph returns DMUS_S_LAST_TOOL if there is no tool other than the output tool waiting to receive the message.

Requirements

Header: Declared in dmusici.h.

See Also

DirectMusic Tools

IDirectMusicInstrument8

The **IDirectMusicInstrument8** interface represents an individual instrument from a DLS collection.

IDirectMusicInstrument8 is a type definition for **IDirectMusicInstrument**. The two interface names are interchangeable.

The only way to create a DirectMusicInstrument object to download an instrument is to first create a DirectMusicCollection object, and then call the **IDirectMusicCollection8::GetInstrument** method. **GetInstrument** creates a DirectMusicInstrument object and returns its **IDirectMusicInstrument8** interface pointer.

To download the instrument, pass its interface pointer to the **IDirectMusicPort8::DownloadInstrument** or the **IDirectMusicPerformance8::DownloadInstrument** method. If the method succeeds, it returns a pointer to an **IDirectMusicDownloadedInstrument8** interface, which is used only to unload the instrument.

The methods of **IDirectMusicInstrument8** operate only on an instrument that has not been downloaded. Any instances of the instrument that have been downloaded to a port are not affected by the **IDirectMusicInstrument8::GetPatch** and **IDirectMusicInstrument8::SetPatch** methods.

The interface has the following methods:

IDirectMusicInstrument8	**GetPatch**
	SetPatch

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTMUSICINTRUMENT8** type is defined as a pointer to this interface.

```
typedef IDirectMusicInstrument8 *LPDIRECTMUSICINSTRUMENT8;
```

Requirements

Header: Declared in dmusicc.h.

IDirectMusicInstrument8::GetPatch

Retrieves the patch number for the instrument. The patch number is an address composed of the MSB and LSB bank selects and the MIDI patch (program change) number. An optional flag bit indicates that the instrument is a drum, rather than a melodic instrument.

```
HRESULT GetPatch(
  DWORD* pdwPatch
);
```

Parameters

pdwPatch
Address of a variable that receives the patch number.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_POINTER

Remarks

The patch number returned at *pdwPatch* describes the full patch address, including the MIDI parameters for MSB and LSB bank select. In addition, the high bit is set if the instrument is specifically a drum kit, intended to be played on MIDI channel 10. This is a special tag for DLS Level 1, because DLS Level 1 always plays drums on MIDI channel 10. For more information, see MIDI Channel Messages.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicInstrument8::SetPatch

Sets the patch number for the instrument. Although each instrument in a DLS collection has a predefined patch number, the patch number can be reassigned after the **IDirectMusicCollection8::GetInstrument** method has been used to retrieve the

instrument from the collection. For more information on DirectMusic patch numbers, see **IDirectMusicInstrument8::GetPatch**.

```
HRESULT SetPatch(
  DWORD dwPatch
);
```

Parameters

dwPatch
New patch number to assign to instrument.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
DMUS_E_INVALIDPATCH

Remarks

The following code example gets an instrument from a collection, remaps its MSB bank select to a different bank, and then downloads the instrument.

```
HRESULT myRemappedDownload(
    IDirectMusicCollection8 *pCollection,
    IDirectMusicPort8 *pPort,
    IDirectMusicDownloadedInstrument8 **ppDLInstrument,
    BYTE bMSB,      // Requested MIDI MSB for patch bank select
    DWORD dwPatch)   // Requested patch

{
    HRESULT hr;
    IDirectMusicInstrument8* pInstrument;
    hr = pCollection->GetInstrument(dwPatch, &pInstrument);
    if (SUCCEEDED(hr))
    {
        dwPatch &= 0xFF00FFFF;  // Clear MSB.
        dwPatch |= bMSB << 16;  // Insert new MSB value.
        pInstrument->SetPatch(dwPatch);
        hr = pPort->DownloadInstrument(pInstrument,
                ppDLInstrument,
                NULL, 0);       // Download all regions.
        pInstrument->Release();
    }
```

```
    return hr;
}
```

Requirements

Header: Declared in dmusicc.h.

IDirectMusicLoader8

The **IDirectMusicLoader8** interface is used for finding, enumerating, caching, and loading objects. For an overview, see Loading Audio Data.

This interface supersedes **IDirectMusicLoader** and adds support for garbage collection.

The methods of the **IDirectMusicLoader8** interface can be organized into the following groups:

Searching	**EnumObject**
	ScanDirectory
	SetSearchDirectory
Cache and memory management	**CacheObject**
	ClearCache
	CollectGarbage
	EnableCache
	ReleaseObject
	ReleaseObjectByUnknown
Object loading	**GetObject**
	LoadObjectFromFile
	SetObject

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDMUS_LOADER** type is defined as a pointer to the **IDirectMusicLoader** interface.

```
typedef IDirectMusicLoader __RPC_FAR *LPDMUS_LOADER;
```

The following table lists the standard types of loadable objects, together with their class identifiers (the *rguidClass* parameter of various methods that deal with objects) and the usual file name extension.

Object type	Class	Extension
Audiopath	CLSID_DirectMusicAudioPathConfig	aud
Band	CLSID_DirectMusicBand	bnd
Container	CLSID_DirectMusicContainer	con
DLS collection	CLSID_DirectMusicCollection	dls
Chordmap	CLSID_DirectMusicChordMap	cdm
Segment	CLSID_DirectMusicSegment	sgt
Script	CLSID_DirectMusicScript	spt
Song*	CLSID_DirectMusicSong	sng
Style	CLSID_DirectMusicStyle	sty
Template	CLSID_DirectMusicSegment	tpl
Toolgraph	CLSID_DirectMusicGraph	tgr
Wave	CLSID_DirectSoundWave	wav

*** Not implemented in DirectX 8.0.**

Requirements

Header: Declared in dmusici.h.

IDirectMusicLoader8::CacheObject

Tells the loader to keep a reference to the object. This guarantees that the object is not loaded twice.

```
HRESULT CacheObject(
  IDirectMusicObject * pObject
);
```

Parameters

pObject

Address of the **IDirectMusicObject** interface of the object to cache. Use **QueryInterface** to obtain **IDirectMusicObject8**.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the object is already cached.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_LOADER_OBJECTNOTFOUND

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::EnableCache, **IDirectMusicLoader8::ClearCache**, **IDirectMusicLoader8::ReleaseObject**, Cache Management

IDirectMusicLoader8::ClearCache

Tells the loader to release all references to a particular type of object.

```
HRESULT ClearCache(
  REFGUID rguidClass
);
```

Parameters

rguidClass
Reference to (C++) or address of (C) the identifier of the class of objects to clear, or GUID_DirectMusicAllTypes to clear all types. For a list of standard loadable classes, see **IDirectMusicLoader8**.

Return Values

The method returns S_OK.

Remarks

This method clears all objects that are currently being held, but does not turn off caching. Use the **IDirectMusicLoader8::EnableCache** method to turn off automatic caching.

To clear a single object from the cache, call the **IDirectMusicLoader8::ReleaseObject** method.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::CacheObject, Cache Management

IDirectMusicLoader8::CollectGarbage

Removes from the cache objects that are no longer in use.

```
CollectGarbage();
```

Parameters

None.

Return Values

None.

Remarks

When an application calls **IDirectMusicLoader8::ReleaseObject** on an object, that object is removed from the cache, and any objects it references become candidates for removal. **IDirectMusicLoader8::CollectGarbage** finds cached objects that are no longer being used by other objects, removes them from the cache, and releases them from memory.

Requirements

Header: Declared in dmusici.h.

See Also

Garbage Collection

IDirectMusicLoader8::EnableCache

Tells the loader to enable or disable automatic caching of all objects it loads. By default, caching is enabled for all classes.

```
HRESULT EnableCache(
  REFGUID rguidClass,
  BOOL fEnable
);
```

Parameters

rguidClass

Reference to (C++) or address of (C) the identifier of the class of objects to cache, or GUID_DirectMusicAllTypes to cache all types. For a list of standard loadable classes, see **IDirectMusicLoader8**.

fEnable

TRUE to enable caching; FALSE to clear and disable.

Return Values

The method returns S_OK if the cache state is changed, or S_FALSE if the cache is already in the desired state.

Remarks

To clear the cache without disabling caching, call the **IDirectMusicLoader8::ClearCache** method.

The following code example disables caching only for segment objects so that they do not stay in memory after the application releases them. Other objects that should be shared, such as styles, chordmaps, and DLS collections, continue to be cached. The first call to **EnableCache** would normally be unnecessary because caching is enabled for all objects by default.

```
void myPrepareLoader(IDirectMusicLoader8 *pILoader)

{
    pILoader->EnableCache(GUID_DirectMusicAllTypes, TRUE);
    pILoader->EnableCache(CLSID_DirectMusicSegment, FALSE);
}
```

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::CacheObject, **IDirectMusicLoader8::ClearCache**, Cache Management

IDirectMusicLoader8::EnumObject

Enumerates all available objects of the requested type. Objects are available if they have been loaded or if **IDirectMusicLoader8::ScanDirectory** has been called on the search directory.

```
HRESULT EnumObject(
  REFGUID rguidClass,
  DWORD dwIndex,
  LPDMUS_OBJECTDESC pDesc
);
```

Parameters

rguidClass
Reference to (C++) or address of (C) the identifier for the class of objects to view. For a list of standard loadable classes, see **IDirectMusicLoader8**.

dwIndex
Index into the list. Typically, starts with 0 and increments.

pDesc
Address of a **DMUS_OBJECTDESC** structure to be filled with data about the object.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if *dwIndex* is past the end of the list.

Remarks

For an example of the use of this method, see Enumerating Objects.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::ScanDirectory

IDirectMusicLoader8::GetObject

Retrieves the specified object from a file or resource and returns the desired interface.

```
HRESULT GetObject(
  LPDMUS_OBJECTDESC pDesc,
  REFIID riid,
  LPVOID FAR * ppv
);
```

Parameters

pDesc
Address of a **DMUS_OBJECTDESC** structure describing the object.

riid
Unique identifier of the interface. See the IID defines in Dmusici.h. All the standard interfaces have a defined identifier consisting of "IID_" plus the name

of the interface. For example, the identifier of **IDirectMusicTrack8** is IID_IDirectMusicTrack8.

ppv

Address of a variable that receives a pointer to the desired interface of the object.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_PARTIALLOAD.

DMUS_S_PARTIALLOAD is returned if any referenced object cannot be found, such as a style referenced in a segment. The loader might fail to find the style if it is referenced by name but **IDirectMusicLoader8::ScanDirectory** has not been called for styles. DMUS_S_PARTIALLOAD might also mean that the default instrument collection file, Gm.dls, is not available.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG
E_OUTOFMEMORY
E_POINTER
DMUS_E_LOADER_NOCLASSID
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED
REGDB_E_CLASSNOTREG

Remarks

For file objects, it is simpler to use the **IDirectMusicLoader8::LoadObjectFromFile** method.

DirectMusic does not support loading from URLs. If the **dwValidData** member of the **DMUS_OBJECTDESC** structure contains DMUS_OBJ_URL, the method returns DMUS_E_LOADER_FORMATNOTSUPPORTED.

The method does not require that all valid members of the **DMUS_OBJECTDESC** structure match before retrieving an object. It searches in the following order:

1. DMUS_OBJ_OBJECT
2. DMUS_OBJ_MEMORY
3. DMUS_OBJ_FILENAME and DMUS_OBJ_FULLPATH
4. DMUS_OBJ_NAME and DMUS_OBJ_CATEGORY
5. DMUS_OBJ_NAME
6. DMUS_OBJ_FILENAME

In other words, the highest priority goes to a unique GUID, followed by a resource, followed by the full file path name, followed by an internal name plus category, followed by an internal name, followed by a local file name.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::ReleaseObject, **IDirectMusicLoader8::ScanDirectory**, **IDirectMusicLoader8::LoadObjectFromFile**

IDirectMusicLoader8::LoadObjectFromFile

Retrieves the specified object from a file and returns the desired interface. This method can be used instead of **IDirectMusicLoader8::GetObject** when the object is in a file.

```
HRESULT LoadObjectFromFile(
  REFGUID rguidClassID,
  REFIID iidInterfaceID,
  WCHAR *pwzFilePath,
  void ** ppObject
);
```

Parameters

rguidClassID
Unique identifier for the class of object. For a list of standard loadable classes, see **IDirectMusicLoader8**.

iidInterfaceID
Unique identifier of the interface. See the IID defines in Dmusici.h. All the standard interfaces have a defined identifier consisting of "IID_" plus the name of the interface. For example, the identifier of **IDirectMusicTrack8** is IID_IDirectMusicTrack8.

pwzFilePath
Name of the file that contains the object. The path can be fully qualified or relative to the search directory.

ppObject
Address of a variable that receives a pointer to the desired interface of the object.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_PARTIALLOAD.

If it fails, the method can return one of the following error values.

E_FAIL
E_INVALIDARG
E_OUTOFMEMORY
E_POINTER
DMUS_E_LOADER_NOCLASSID
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED
REGDB_E_CLASSNOTREG

Remarks

DMUS_S_PARTIALLOAD is returned if any referenced object cannot be found, such as a style referenced in a segment. The loader might fail to find the style if it is referenced by name but **IDirectMusicLoader8::ScanDirectory** has not been called for styles. DMUS_S_PARTIALLOAD might also mean that the default instrument collection file, Gm.dls, is not available.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::GetObject, **IDirectMusicLoader8::SetSearchDirectory**

IDirectMusicLoader8::ReleaseObject

Releases the loader's reference to the object.

```
HRESULT ReleaseObject(
  IDirectMusicObject * pObject
);
```

Parameters

pObject
IDirectMusicObject8 interface pointer of the object to release.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the object has already been released or cannot be found in the cache.

If it fails, the method can return E_POINTER.

Remarks

ReleaseObject is the reciprocal of **IDirectMusicLoader8::CacheObject**.

Objects can be cached explicitly by using the **CacheObject** method, or automatically by using the **IDirectMusicLoader8::EnableCache** method.

To tell the loader to flush all objects of a particular type, call the **IDirectMusicLoader8::ClearCache** method.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::ReleaseObjectByUnknown, **DirectMusicLoader8::GetObject**, Cache Management

IDirectMusicLoader8::ReleaseObjectByUnknown

Releases the loader's reference to the object. This method is similar to **IDirectMusicLoader8::ReleaseObject** and is suitable for releasing objects for which the **IDirectMusicObject8** interface is not readily available.

```
HRESULT ReleaseObject(
  IUnknown * pObject
);
```

Parameters

pObject
Address of the **IUnknown** interface pointer of the object to release.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the object has already been released or cannot be found in the cache.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::GetObject, Cache Management

IDirectMusicLoader8::ScanDirectory

Searches a directory or disk for all files of a requested class type and file name extension. For each file found, it calls the **IDirectMusicObject8::ParseDescriptor** method to extract the GUID and name of the object. This information is stored in an internal database. After a directory has been scanned, all files of the requested type become available for enumeration through the **IDirectMusicLoader8::EnumObject** method; in addition, an object can be retrieved by using **IDirectMusicLoader8::GetObject**, even without a file name.

```
HRESULT ScanDirectory(
  REFGUID rguidClass,
  WCHAR* pwzFileExtension,
  WCHAR* pwzScanFileName
);
```

Parameters

rguidClass
Reference to (C++) or address of (C) the identifier of the class of objects. For a list of standard loadable classes, see **IDirectMusicLoader8**.

pwzFileExtension
File name extension for the type of file to look for. Use L"*" to look in files with any or no extension. See Remarks.

pwzScanFileName
Optional storage file to store and retrieve cached file information. This file is created by the first call to **ScanDirectory** and used by subsequent calls. Pass NULL if a cache file is not wanted.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if no files were found.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
E_FAIL
E_OUTOFMEMORY
E_POINTER

REGDB_E_CLASSNOTREG

Remarks

The **IDirectMusicLoader8::SetSearchDirectory** method must be called first to set the location to search.

The scanned information can be stored in a cache file defined by *pwzScanFileName*. After it has been so stored, subsequent calls to **ScanDirectory** are much quicker because only files that have changed are scanned (the cache file stores the file size and date for each object, so it can tell if a file has changed).

GUID_DirectMusicAllTypes is not a valid value for *rguidClass*.

If the file type has more than one possible extension, call **ScanDirectory** once for each file name extension.

Requirements

Header: Declared in dmusici.h.

See Also

Scanning a Directory for Objects

IDirectMusicLoader8::SetObject

Tells the loader where to find an object when it is later referenced by another object being loaded, and adds attributes to an object so that it can be referred to by those attributes. For an overview, see Setting Objects.

```
HRESULT SetObject(
  LPDMUS_OBJECTDESC pDesc
);
```

Parameters

pDesc

Address of a **DMUS_OBJECTDESC** structure describing the object. On entry, this structure contains any information the application has about the object. On return, it can contain additional information.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG
E_OUTOFMEMORY
E_POINTER
DMUS_E_LOADER_NOCLASSID
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED
REGDB_E_CLASSNOTREG

Remarks

This method can be used to set attributes that are not currently valid for an object. For example, you can supply a value in the **wszName** member of the **DMUS_OBJECTDESC** structure to assign an internal name to an unnamed object, such as a segment based on a MIDI file. However, it cannot be used to change existing attributes. Most authored segments, for example, already have names, and these cannot be changed by the application.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::GetObject

IDirectMusicLoader8::SetSearchDirectory

Sets a search path for finding object files. The search path can be set for one object file type or for all files.

```
HRESULT SetSearchDirectory(
  REFGUID rguidClass,
  WCHAR* pwszPath,
  BOOL fClear
);
```

Parameters

rguidClass

Reference to (C++) or address of (C) the identifier of the class of objects that the call pertains to. GUID_DirectMusicAllTypes specifies all objects. For a list of standard loadable classes, see **IDirectMusicLoader8**.

pwszPath

File path for directory. Must be a valid directory and must be less than MAX_PATH in length. The path, if not fully qualified, is relative to the current directory when **IDirectMusicLoader8::ScanDirectory** is called.

fClear

If TRUE, clears all information about objects before setting the directory. This prevents accessing objects from the previous directory that might have the same name. However, objects are not removed from the cache.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the search directory is already set to *pwszPath*.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY

E_POINTER

DMUS_E_LOADER_BADPATH

Remarks

After a search path is set, the loader does not need a full path every time it is given an object to load by file name. This enables objects that refer to other objects to find them by file name without knowing the full path.

When this method has been called, the loader expects the **wszFileName** member of the **DMUS_OBJECTDESC** structure to contain only a file name or a path relative to the search directory, unless the DMUS_OBJ_FULLPATH flag is set in the **dwValidData** member.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader8::ScanDirectory, Setting the Loader's Search Directory

IDirectMusicObject8

All DirectMusic objects that can be loaded from a file support the **IDirectMusicObject8** interface so that they can work with the DirectMusic loader.

New types of objects need to implement this interface. For more information, see Custom Loading.

Most applications do not use the methods of this interface directly. However, **IDirectMusicObject8::GetDescriptor** can be used to query an object for information, including its name, GUID, file path, and version.

The **IDirectMusicObject8** interface is usually obtained by calling another interface's **QueryInterface** method. It cannot be obtained by using **CoCreateInstance**. The interface is also returned by **IDirectMusicContainer8::EnumObject**.

IDirectMusicObject8 is a define for **IDirectMusicObject**. The two interface names are interchangeable.

The **IDirectMusicObject8** interface has the following methods:

Descriptor	**GetDescriptor**
	ParseDescriptor
	SetDescriptor

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDMUS_OBJECT** type is defined as a pointer to the **IDirectMusicObject** interface.

```
typedef IDirectMusicObject __RPC_FAR *LPDMUS_OBJECT;
```

Requirements

Header: Declared in dmusici.h.

See Also

Loading Audio Data, Custom Loading

IDirectMusicObject8::GetDescriptor

Retrieves the object's internal description.

```
HRESULT GetDescriptor(
  LPDMUS_OBJECTDESC pDesc
);
```

Parameters

pDesc

Address of a **DMUS_OBJECTDESC** structure to be filled with data about the object. Depending on the implementation of the object and how it was loaded from a file, some or all of the standard parameters are filled by **GetDescriptor**. Check the flags in the **dwValidData** member to ascertain which other members are valid.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_GARBAGE_COLLECTED. See Garbage Collection.

If it fails, the method can return E_POINTER.

Remarks

For an example, see Getting Object Descriptors.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicObject8::SetDescriptor

IDirectMusicObject8::ParseDescriptor

Given a file stream, the **IDirectMusicObject8::ParseDescriptor** method scans the file for data that it can store in the **DMUS_OBJECTDESC** structure. All members that are supplied are marked with the appropriate flags in **dwValidData**.

This method is primarily used by the loader when scanning a directory for objects, and is not normally used directly by an application. However, if an application implements an object type in DirectMusic, it should support this method.

```
HRESULT ParseDescriptor(
  LPSTREAM pStream,
  LPDMUS_OBJECTDESC pDesc
);
```

Parameters

pStream

Stream source for the file.

pDesc

Address of a **DMUS_OBJECTDESC** structure that receives data about the file.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_GARBAGE_COLLECTED. See Garbage Collection.

If it fails, the method can return one of the following error values:

DMUS_E_CHUNKNOTFOUND
DMUS_E_INVALID_BAND
DMUS_E_INVALIDFILE
DMUS_E_NOTADLSCOL
E_FAIL
E_OUTOFMEMORY
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicObject8::SetDescriptor

IDirectMusicObject8::SetDescriptor

Sets some or all members of the object's internal description.

This method is primarily used by the loader when creating an object, and is not normally used directly by an application. However, if an application implements an object type in DirectMusic, it should support this method.

```
HRESULT SetDescriptor(
  LPDMUS_OBJECTDESC pDesc
);
```

Parameters

pDesc
Address of a **DMUS_OBJECTDESC** structure that receives data about the object. Data is copied to all members that are enabled in the **dwValidData** member

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The descriptor was set.
S_FALSE	See Remarks.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_POINTER

Remarks

Applications do not normally call this method on standard objects. Although it is possible to change the object descriptor returned by **IDirectMusicObject8::GetDescriptor**, the new description cannot successfully be passed to the **IDirectMusicLoader8::GetObject** method. For example, you could change the name of an object, but **GetObject** will still find the object only under its original name, because it relies on the object's own implementation of **SetDescriptor**.

Members that are not copied keep their previous values. For example, an object might already have its name and GUID stored internally. A call to its **SetDescriptor** method with a new name and file path (and DMUS_OBJ_NAME | DMUS_OBJ_FILENAME in the **dwValidData** member) would replace the name, supply a file name, and leave the GUID as it is.

If the object is unable to set one or more members, it sets the members that it does support, clears the flags in **dwValidData** that it does not support, and returns S_FALSE. An application-defined object should support at least DMUS_OBJ_NAME and DMUS_OBJ_OBJECT.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicObject8::ParseDescriptor, **IDirectMusicObject8::GetDescriptor**

IDirectMusicPatternTrack8

The **IDirectMusicPatternTrack8** interface represents a track that contains a single pattern. A pattern track is similar to a sequence track, but because it contains music values rather than fixed notes, it responds to chord changes.

You can obtain this interface by passing IID_DirectMusicPatternTrack to the **IDirectMusicTrack8::QueryInterface** method of the track.

The **IDirectMusicPatternTrack8** interface has the following methods:

Creating a segment	**CreateSegment**
Setting the pattern	**SetPatternByName**
Setting the variation	**SetVariation**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicPatternTrack8::CreateSegment

Creates a segment containing the pattern track.

```
HRESULT CreateSegment(
  IDirectMusicStyle* pStyle,
  IDirectMusicSegment** ppSegment
);
```

Parameters

pStyle
Style to use in creating the segment.

ppSegment
Address of a variable that receives an **IDirectMusicSegment** interface pointer for the created segment. Use QueryInterface to obtain **IDirectMusicSegment8**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

DMUS_E_NOT_INIT
E_OUTOFMEMORY

Remarks

A pattern track is usually obtained from a segment object loaded from a file or resource, in which case the application already has the **IDirectMusicSegment** interface. This method is used for creating a segment when the pattern track object has been created by using **CoCreateInstance**.

Requirements

Header: Declared in dmusici.h.

IDirectMusicPatternTrack8::SetPatternByName

Sets the pattern to be played by the track. The pattern comes from a style.

```
HRESULT SetPatternByName(
  IDirectMusicSegmentState* pSegState,
  WCHAR* wszName,
  IDirectMusicStyle* pStyle,
  DWORD dwPatternType,
  DWORD* pdwLength
);
```

Parameters

pSegState
Address of the **IDirectMusicSegmentState8** interface of the segment state that is playing the track.

wszName
Name of the pattern to set. The name can be obtained by using **IDirectMusicStyle8::EnumPattern**.

pStyle
Address of the **IDirectMusicStyle** or **IDirectMusicStyle8** interface of the style containing the pattern.

dwPatternType
One of the **DMUS_STYLET_TYPES** enumeration that specifies the type of pattern.

pdwLength
Address of a variable that receives the length of the pattern, in music time ticks.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

DMUS_E_NOT_FOUND
DMUS_E_NOT_INIT
E_OUTOFMEMORY
E_POINTER

Requirements

Header: Declared in dmusici.h.

IDirectMusicPatternTrack8::SetVariation

Sets the variations to be played by a part in the track.

```
HRESULT SetVariation(
  IDirectMusicSegmentState* pSegState,
  DWORD dwVariationFlags,
  DWORD dwPart
);
```

Parameters

pSegState
Address of the **IDirectMusicSegmentState8** interface of the segment state that is playing the track.

dwVariationFlags
Bitmask where a bit is set for each variation that is to be played.

dwPart
Identifier for the part containing the variations. This is the number assigned to the part in the music-authoring application, and is equivalent to the PChannel.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_POINTER

Remarks

Variations can be set for only one part at a time. Each time this method is called, it overrides previous calls.

The following example code plays variations 16 and 32 on performance channel 1.

```
// pPattern is an IDirectMusicPatternTrack8 pointer.
// pSegmentState is an IDirectMusicSegmentState8 pointer.

#define VARIATION(v)  (1 << ((v) - 1))

HRESULT hr = pPattern->SetVariation(
   pSegmentState, VARIATION(32) | VARIATION(16), 1);
```

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8

The **IDirectMusicPerformance8** interface is the overall manager of music playback. It is used for adding and removing ports, mapping performance channels to ports, playing segments, dispatching messages and routing them through tools, requesting and receiving event notification, and setting and retrieving music parameters. It also has several methods for getting information about timing and for converting time and music values from one system to another.

If an application needs two complete sets of music playing at the same time, it can do so by creating more than one performance. Separate performances obey separate tempo maps, and so play completely asynchronously, whereas all segments within one performance play in lock step.

IDirectMusicPerformance8 supersedes the **IDirectMusicPerformance** interface and adds new methods.

The methods of **IDirectMusicPerformance8** can be organized into the following groups:

Audiopath	**CreateAudioPath**
	CreateStandardAudioPath
	GetDefaultAudioPath
	SetDefaultAudioPath
Channels	**AssignPChannel**
	AssignPChannelBlock
	PChannelInfo
Instruments	**DownloadInstrument**
Messages	**AllocPMsg**
	ClonePMsg
	FreePMsg
	SendPMsg
MIDI conversion	**MIDIToMusic**
	MusicToMIDI

Notification	**AddNotificationType**
	GetNotificationPMsg
	RemoveNotificationType
	SetNotificationHandle
Parameters	**GetGlobalParam**
	GetParam
	GetParamEx
	SetGlobalParam
	SetParam
Playback	**GetSegmentState**
	IsPlaying
	PlaySegment
	PlaySegmentEx
	Stop
	StopEx
Ports	**AddPort**
	RemovePort
Timing	**AdjustTime**
	GetBumperLength
	GetLatencyTime
	GetPrepareTime
	GetQueueTime
	GetResolvedTime
	GetTime
	MusicToReferenceTime
	ReferenceToMusicTime
	RhythmToTime
	SetBumperLength
	SetPrepareTime
	TimeToRhythm
Tools	**GetGraph**
	SetGraph
Miscellaneous	**CloseDown**
	Init
	InitAudio
	Invalidate

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::AddNotificationType

Adds a notification type to the performance. All segments and tracks are automatically updated with the new notification by calling their **AddNotificationType** methods.

```
HRESULT AddNotificationType(
  REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType

Reference to (C++) or address of (C) the identifier of the notification type to add. For the defined types, see **DMUS_NOTIFICATION_PMSG**. Applications can also define their own types for custom tracks.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY

E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::RemoveNotificationType, **IDirectMusicSegment8::AddNotificationType**, **IDirectMusicTrack8::AddNotificationType**, Notification and Event Handling

IDirectMusicPerformance8::AddPort

Assigns a port to the performance. This method is valid only for performances that do not use audiopaths; that is, performances initialized by using **IDirectMusicPerformance8::Init**.

```
HRESULT AddPort(
  IDirectMusicPort* pPort
);
```

Parameters

pPort
Address of a variable that contains the port to add. If NULL, the default port is added. See Remarks.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
DMUS_E_CANNOT_OPEN_PORT
E_OUTOFMEMORY
E_POINTER

Remarks

When the default port is specified by passing NULL in *pPort*, it is assigned one channel group. If no performance channels have been set up for any other port, channels 0 through 15 are assigned to MIDI channels 0 through 15.

If *pPort* is not NULL, it must be a port created by the same DirectMusic object that was passed to, or created by, **IDirectMusicPerformance8::Init**. The port must be activated by a call to **IDirectMusicPort8::Activate**, and a block of channels must be assigned by a call to **IDirectMusicPerformance8::AssignPChannelBlock**.

This method creates a reference to **IDirectMusicPort8** that is released by **IDirectMusicPerformance8::RemovePort** or **IDirectMusicPerformance8::CloseDown**. However, if NULL is passed to **AddPort**,

the port cannot be removed by **RemovePort**, because the application has no reference to pass to **RemovePort**.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::RemovePort, Default Port

IDirectMusicPerformance8::AdjustTime

Adjusts the internal performance time forward or backward. This is mostly used to compensate for drift when synchronizing to another source.

```
HRESULT AdjustTime(
  REFERENCE_TIME rtAmount
);
```

Parameters

rtAmount

Amount of time to add or subtract. This can be a number from –10,000,000 through 10,000,000 (–1 second through +1 second).

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_INVALIDARG.

Remarks

The adjusted time is used internally by DirectMusic. It is not reflected in the time retrieved by the **IDirectMusicPerformance8::GetTime** method.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetTime, Timing

IDirectMusicPerformance8::AllocPMsg

Allocates a performance message.

```
HRESULT AllocPMsg(
  ULONG cb,
  DMUS_PMSG** ppPMSG
);
```

Parameters

cb
Size of the message structure. This structure is of a type derived from **DMUS_PMSG**.

ppPMSG
Address of a variable that receives the pointer to the allocated message structure.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_OUTOFMEMORY
E_POINTER

Remarks

The **dwSize** member of the message structure is set to the value of *cb*. Other members are not necessarily initialized to zero, because of internal caching.

After the message is sent by **IDirectMusicPerformance8::SendPMsg**, the application no longer owns the memory and is not responsible for freeing the message. However, a tool can free a message within its **IDirectMusicTool8::Flush** or its **IDirectMusicTool8::ProcessPMsg** method. Applications are also responsible for freeing notification messages.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::FreePMsg, **IDirectMusicPerformance8::SendPMsg**, DirectMusic Messages

IDirectMusicPerformance8::AssignPChannel

Assigns a single performance channel to the performance and maps it to a port, group, and MIDI channel.

This method is not used by applications that route their data through audiopaths.

```
HRESULT AssignPChannel(
  DWORD dwPChannel,
  IDirectMusicPort* pPort,
  DWORD dwGroup,
  DWORD dwMChannel
);
```

Parameters

dwPChannel
Performance channel to assign.

pPort
Address of a variable that contains the port to which the channel is assigned.

dwGroup
Channel group on the port.

dwMChannel
Channel in the group. Must be in the range from 0 through 15.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE (see Remarks).

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_POINTER

Remarks

The method returns S_FALSE if *dwGroup* is out of the range of the port. The channel has been assigned, but the port cannot play this group.

The method returns E_INVALIDARG if *dwMChannel* is out of range or the port has not been added to the performance through a call to the **IDirectMusicPerformance8::AddPort** method.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::AssignPChannelBlock, **IDirectMusicPerformance8::PChannelInfo**, Channels

IDirectMusicPerformance8::AssignPChannelBlock

Assigns a block of 16 performance channels to the performance and maps them to a port and a channel group. This method is valid only for performances that do not use audiopaths; that is, performances initialized by using **IDirectMusicPerformance8::Init**.

```
HRESULT AssignPChannelBlock(
  DWORD dwBlockNum,
  IDirectMusicPort* pPort,
  DWORD dwGroup
);
```

Parameters

dwBlockNum
Block number, in which 0 represents channels 0 through 15, 1 represents channels 16 through 31, and so on.

pPort
Address of a variable that contains the port to which the channels are assigned.

dwGroup
Channel group on the port. Must be 1 or greater.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE (see Remarks).

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_POINTER

Remarks

This method must be called when a port has been added to a performance, except when the default port has been added by passing NULL to **IDirectMusicPerformance8::AddPort**.

The method returns S_FALSE if *dwGroup* is out of the range of the port. The channels have been assigned, but the port cannot play this group.

The method returns E_INVALIDARG if the port has not been added to the performance through a call to the **IDirectMusicPerformance8::AddPort** method.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::AssignPChannel, **IDirectMusicPerformance8::PChannelInfo**, Channels

IDirectMusicPerformance8::ClonePMsg

Makes a copy of a performance message.

```
HRESULT ClonePMsg(
  DMUS_PMSG* pSourcePMSG,
  DMUS_PMSG** ppCopyPMSG
);
```

Parameters

pSourcePMSG
Message to copy.

ppCopyPMSG
Address of a variable that receives a pointer to the copied message.

Return Values

If it succeeds, the method returns S_OK.

If it fails, the method can return one of the following error values.

E_OUTOFMEMORY
E_POINTER

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::CloseDown

Closes down the performance object. An application that created the performance object and called **IDirectMusicPerformance8::Init** or **IDirectMusicPerformance8::InitAudio** on it must call **CloseDown** before the performance is released.

```
HRESULT CloseDown();
```

Parameters

None.

Return Values

The method returns S_OK.

Remarks

Failure to call **CloseDown** can cause memory leaks or program failures.

CloseDown handles the release of the **IDirectMusic8** interface if this reference was created by **IDirectMusicPerformance8::Init** or **IDirectMusicPerformance8::InitAudio**. If the application explicitly created the DirectMusic object, the application is responsible for releasing the reference.

If the DirectSound object was created in the call to **Init** or **InitAudio** but no reference was returned to the application, **CloseDown** also releases DirectSound and all DirectSound buffers. If your application has obtained any interfaces to DirectSound buffers, it should release them before calling **Closedown**.

If the application created DirectSound explicitly, or obtained a reference form **Init** or **InitAudio**, it is responsible for releasing DirectSound.

The method releases any downloaded instruments that have not been unloaded.

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::CreateAudioPath

Creates an audiopath object that represents the stages in data flow from the performance to DirectSound buffers.

```
HRESULT CreateAudioPath(
  IUnknown *pSourceConfig,
  BOOL fActivate,
  IDirectMusicAudioPath **ppNewPath
);
```

Parameters

pSourceConfig
Address of an interface that represents the audiopath configuration. See Remarks.

fActivate
Boolean value that specifies whether to activate the path on creation.

ppNewPath
Address of a variable that receives an **IDirectMusicAudioPath8** interface pointer for the audiopath.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

E_NOINTERFACE
E_POINTER
DMUS_E_NOT_INIT
DMUS_E_AUDIOPATHS_NOT_VALID
DSERR_BUFFERLOST
E_OUTOFMEMORY

Remarks

The object addressed by *pSourceConfig* can be obtained from a segment by using the **IDirectMusicSegment8::GetAudioPathConfig** method or can be loaded directly from a file.

The method fails with DSERR_BUFFERLOST if any application has initialized DirectSound with the write-primary cooperative level.

If the audiopath configuration specifies a sound device that is not available, the method returns E_NOINTERFACE.

Requirements

Header: Declared in dmusici.h.

See Also

Creating Audiopaths

IDirectMusicPerformance8::CreateStandardAudioPath

Creates an object that represents the stages in data flow from the performance to DirectSound buffers. This method can be used instead of **IDirectMusicPerformance8::CreateAudioPath** to create a basic predefined audiopath rather than one defined in a source file.

```
HRESULT CreateStandardAudioPath(
  DWORD dwType,
  DWORD dwPChannelCount,
  BOOL  fActivate,
  IDirectMusicAudioPath **ppNewPath
);
```

Parameters

dwType

Type of the path. The following values are defined.

DMUS_APATH_DYNAMIC_3D
One bus to a 3-D buffer. Does not send to environmental reverb.

DMUS_APATH_DYNAMIC_MONO
One bus to a mono buffer.

DMUS_APATH_SHARED_STEREOPLUSREVERB
Ordinary music setup with stereo outs and reverb.

DMUS_APATH_DYNAMIC_STEREO
Two buses to a stereo buffer.

For more information on these audiopath types, see Standard Audiopaths.

dwPChannelCount

Number of performance channels in the path.

fActivate

Boolean value that specifies whether to activate the path on creation.

ppNewPath

Address of a variable that receives an **IDirectMusicAudioPath** interface pointer for the audiopath. See **IDirectMusicAudioPath8**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER

DMUS_E_NOT_INIT

DMUS_E_AUDIOPATHS_NOT_VALID

DSERR_BUFFERLOST

E_INVALIDARG

E_OUTOFMEMORY

Remarks

The method fails with DSERR_BUFFERLOST if any application has initialized DirectSound with the write-primary cooperative level.

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::DownloadInstrument

Downloads DLS data for an instrument to a port.

```
HRESULT DownloadInstrument(
  IDirectMusicInstrument* pInst,
  DWORD dwPChannel,
  IDirectMusicDownloadedInstrument** ppDownInst,
  DMUS_NOTERANGE* pNoteRanges,
  DWORD dwNumNoteRanges,
  IDirectMusicPort** ppPort,
  DWORD* pdwGroup,
  DWORD* pdwMChannel
);
```

Parameters

pInst
Pointer to the **IDirectMusicInstrument8** interface of the instrument to download.

dwPChannel
Performance channel to which the instrument is assigned.

ppDownInst
Address of a variable that receives an **IDirectMusicDownloadedInstrument8** pointer to the downloaded instrument.

pNoteRanges
Address of an array of **DMUS_NOTERANGE** structures. Each entry in the array specifies a contiguous range of MIDI note messages to which the instrument must respond. An instrument region is downloaded only if at least one note in that region is specified in the **DMUS_NOTERANGE** structures.

dwNumNoteRanges
Number of **DMUS_NOTERANGE** structures in the array pointed to by *pNoteRanges*. If this value is set to 0, the *pNoteRanges* parameter is ignored, and all regions and wave data for the instrument are downloaded.

ppPort

Address of a variable that receives a pointer to the port to which the instrument was downloaded.

pdwGroup
Address of a variable that receives the group to which the instrument is assigned.

pdwMChannel
Address of a variable that receives the MIDI channel to which the instrument is assigned.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_POINTER

Remarks

Most applications do not need to use this method because instrument downloading is normally handled by bands. See Downloading and Unloading Bands.

The method returns E_INVALIDARG if the performance channel is not assigned to a port.

To prevent loss of resources, unload the instrument by using the **IDirectMusicPort8::UnloadInstrument** method when the instrument is no longer needed.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPort8::DownloadInstrument, **IDirectMusicPort8::UnloadInstrument**, Working with Instruments

IDirectMusicPerformance8::FreePMsg

Frees a message.

```
HRESULT FreePMsg(
  DMUS_PMSG* pPMSG
);
```

Parameters

pPMSG
Address of a variable that contains a message to free. This message must have been allocated using the **IDirectMusicPerformance8::AllocPMsg** method.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_CANNOT_FREE
E_POINTER

Remarks

Most messages are released automatically by the performance after they have been processed, and **IDirectMusicPerformance8::FreePMsg** must not be called on a message that has been sent by using **IDirectMusicPerformance8::SendPMsg**. However, **IDirectMusicPerformance8::FreePMsg** can be used within **IDirectMusicTool8::ProcessPMsg** or **IDirectMusicTool8::Flush** to free a message that is no longer needed. It must also be used to free notification messages.

The method returns DMUS_E_CANNOT_FREE in the following cases:

- If *pPMSG* is not a message allocated by **AllocPMsg**.
- If it is in the performance queue because **IDirectMusicPerformance8::SendPMsg** was called on it. However, applications cannot assume that **FreePMsg** will return DMUS_E_CANNOT_FREE for all sent messages, because **SendPMsg** is not synchronous.
- If it has already been freed.

If there is a value in the **pTool**, **pGraph**, or **punkUser** members (see **DMUS_PMSG**), each referenced object is released.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::AllocPMsg, DirectMusic Messages

IDirectMusicPerformance8::GetBumperLength

Retrieves the amount of time between the time at which messages are placed in the port buffer and the time at which they begin to be processed by the port. For an overview of this topic, see Timing.

```
HRESULT GetBumperLength(
  DWORD* pdwMilliSeconds
);
```

Parameters

pdwMilliSeconds
Address of a variable that contains the amount of preplay time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

The default value is 50 milliseconds.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SetBumperLength

IDirectMusicPerformance8::GetDefaultAudioPath

Retrieves the default audiopath set by **IDirectMusicPerformance8::InitAudio** or **IDirectMusicPerformance8::SetDefaultAudioPath**.

```
HRESULT GetDefaultAudioPath(
  IDirectMusicAudioPath **ppAudioPath
);
```

Parameters

ppAudioPath
Address of a variable that receives the **IDirectMusicAudioPath8** interface pointer of the default audiopath.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_AUDIOPATHS_NOT_VALID
DMUS_E_NOT_INIT

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::GetGlobalParam

Retrieves global values from the performance.

```
HRESULT GetGlobalParam(
  REFGUID rguidType,
  void* pParam,
  DWORD dwSize
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data.

pParam
Pointer to the allocated memory that receives a copy of the data. This must be the correct size, which is constant for each type of data. This parameter contains information that was passed in to the **IDirectMusicPerformance8::SetGlobalParam** method.

dwSize
Size of the data. This is constant for each *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_POINTER

Remarks

If **SetGlobalParam** has never been called for *rguidType*, the parameter might not be in the list of global data being handled by this performance, and the method might return E_INVALIDARG. In other words, do not assume that any parameter has a default value that can be retrieved by using **GetGlobalParam**.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SetGlobalParam,
IDirectMusicPerformance8::GetParam, Performance Parameters

IDirectMusicPerformance8::GetGraph

Retrieves the toolgraph of a performance.

```
HRESULT GetGraph(
  IDirectMusicGraph** ppGraph
);
```

Parameters

ppGraph
Address of a variable that receives a pointer to the toolgraph.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
E_POINTER

Remarks

The reference count of the graph is incremented.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SetGraph, **IDirectMusicSegment8::GetGraph**, **IDirectMusicPerformance8::SendPMsg**

IDirectMusicPerformance8::GetLatencyTime

Retrieves the current latency time. Latency time is the time being heard from the speakers plus the time required to queue and render messages into the **IDirectMusicPort8**. For an overview of this topic, see Timing.

```
HRESULT GetLatencyTime(
  REFERENCE_TIME * prtTime
);
```

Parameters

prtTime
Address of a variable that receives the current latency time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_NO_MASTER_CLOCK

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::GetNotificationPMsg

Retrieves a pending notification message.

```
HRESULT GetNotificationPMsg(
  DMUS_NOTIFICATION_PMSG** ppNotificationPMsg
);
```

Parameters

ppNotificationPMsg
Address of a variable that receives a pointer to a **DMUS_NOTIFICATION_PMSG** structure. The application retrieving this message is responsible for calling **IDirectMusicPerformance8::FreePMsg** on it.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if there are no more notification events to return.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

Notification and Event Handling

IDirectMusicPerformance8::GetParam

Retrieves data from a track inside the control segment.

```
HRESULT GetParam(
  REFGUID rguidType,
  DWORD dwGroupBits,
  DWORD dwIndex,
  MUSIC_TIME mtTime,
  MUSIC_TIME* pmtNext,
  void* pParam
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data to obtain. See Standard Track Parameters.

dwGroupBits

Group of the track from which to obtain the data (see Remarks). Set this value to 0xFFFFFFFF for all groups.

dwIndex
Index of the track in the group from which to obtain the data.

mtTime
Time from which to obtain the data, in performance time.

pmtNext
Address of a variable that receives the time (relative to *mtTime*) until which the data is valid. If this returns a value of 0, either the data is always valid, or it is not known when it might become invalid. If this information is not needed, *pmtNext* can be set to NULL. See Remarks.

pParam
Address of an allocated structure in which the data is to be returned. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_GET_UNSUPPORTED
DMUS_E_NO_MASTER_CLOCK
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND
E_POINTER

Remarks

Normally, the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. For more information, see Control Segments.

The data returned in **pParam* can become invalid before the time returned in **pmtNext* if another control segment is cued.

Each track belongs to one or more groups, and each group is represented by a bit in *dwGroupBits*. For more information, see **IDirectMusicSegment8::InsertTrack** and Identifying the Track.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SetParam, **IDirectMusicSegment8::GetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicPerformance8::SetGlobalParam**, **IDirectMusicPerformance8::GetTime**, **IDirectMusicPerformance8::GetParamEx**, Performance Parameters

IDirectMusicPerformance8::GetParamEx

Retrieves data from a track. This method is similar to **IDirectMusicPerformance8::GetParam** but adds support for self-controlling segments. It is used chiefly by tools.

```
HRESULT GetParamEx(
  REFGUID rguidType,
  DWORD dwTrackID,
  DWORD dwGroupBits,
  DWORD dwIndex,
  MUSIC_TIME mtTime,
  MUSIC_TIME* pmtNext,
  void* pParam
);
```

Parameters

rguidType

Reference to (C++) or address of (C) the identifier of the type of data to obtain. See Standard Track Parameters.

dwTrackID

Unique identifier of a track within the segment state from which the parameter is to be obtained. Every performance message that originates from a track carries an identifier of the track instance that generated the message. This identifier is kept in the **dwVirtualTrackID** member of the **DMUS_PMSG** part of the message structure. When this value is passed to **GetParamEx**, the method is able to determine whether the track is self-controlling, in which case it gets its data from another track in the same segment rather than in the control segment.

dwGroupBits

Group that the track is in (see Remarks). Set this value to 0xFFFFFFFF for all groups.

dwIndex

Index of the track in the group.

mtTime

Time from which to obtain the data, in performance time.

pmtNext

Address of a variable that receives the time (relative to *mtTime*) until which the data is valid. If this returns a value of 0, either the data is always valid, or it is not known when it might become invalid. If this information is not needed, *pmtNext* can be set to NULL. See Remarks.

pParam

Address of an allocated structure in which the data is to be returned. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_GET_UNSUPPORTED
DMUS_E_NO_MASTER_CLOCK
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND
E_POINTER

Remarks

Each track belongs to one or more groups, and each group is represented by a bit in *dwGroupBits*. For more information, see **IDirectMusicSegment8::InsertTrack** and Identifying the Track.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SetParam, **IDirectMusicSegment8::GetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicPerformance8::SetGlobalParam**, **IDirectMusicPerformance8::GetTime**, Performance Parameters

IDirectMusicPerformance8::GetPrepareTime

Retrieves the interval between the time when messages are sent by tracks and the time when the sound is heard. This interval allows sufficient time for the message to be processed by tools.

HRESULT GetPrepareTime(

```
  DWORD* pdwMilliSeconds
);
```

Parameters

pdwMilliSeconds
Address of a variable that receives the amount of prepare time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

The default value is 1000 milliseconds.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SetPrepareTime, Prepare Time

IDirectMusicPerformance8::GetQueueTime

Retrieves the current flush time, which is the earliest time in the queue at which messages can be flushed. Messages that have time stamps earlier than this time have already been sent to the port and cannot be invalidated.

```
HRESULT GetQueueTime(
  REFERENCE_TIME * prtTime
);
```

Parameters

prtTime
Address of a variable that receives the current flush time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_NO_MASTER_CLOCK

Requirements

Header: Declared in dmusici.h.

See Also

Latency and Bumper Time

IDirectMusicPerformance8::GetResolvedTime

Resolves a given time to a given boundary.

```
HRESULT GetResolvedTime(
  REFERENCE_TIME rtTime,
  REFERENCE_TIME* prtResolved,
  DWORD dwTimeResolveFlags
);
```

Parameters

rtTime
Time to resolve. If this is less than the current time, the current time is used.

prtResolved
Address of a variable that receives the resolved time.

dwTimeResolveFlags
One or more **DMUS_TIME_RESOLVE_FLAGS** describing the resolution desired.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

Timing

IDirectMusicPerformance8::GetSegmentState

Retrieves the currently playing primary segment state or the primary segment state that is playing at a given time.

```
HRESULT GetSegmentState(
  IDirectMusicSegmentState ** ppSegmentState,
  MUSIC_TIME mtTime
);
```

Parameters

ppSegmentState
Address of a variable that receives a pointer to the segment state. The caller is responsible for calling **Release** on this pointer.

mtTime
Time for which the segment state is to be retrieved.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
E_POINTER

Remarks

To get the currently playing segment state, pass the time returned by the **IDirectMusicPerformance8::GetTime** method. Because of latency, the currently playing segment state is not necessarily the one being heard.

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::GetTime

Retrieves the current time of the performance.

```
HRESULT GetTime(
```

```
  REFERENCE_TIME* prtNow,
  MUSIC_TIME* pmtNow
);
```

Parameters

prtNow
Address of a variable that receives the current time in **REFERENCE_TIME** format. Can be NULL.

pmtNow
Address of a variable that receives the current time in **MUSIC_TIME** format. Can be NULL.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NO_MASTER_CLOCK
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

Timing

IDirectMusicPerformance8::Init

Associates the performance with a DirectMusic object and a DirectSound object. If the application is not using audiopaths, this method must be called before the performance can play.

For applications that use audiopaths, this method has been superseded by **IDirectMusicPerformance8::InitAudio**.

```
HRESULT Init(
  IDirectMusic** ppDirectMusic,
  LPDIRECTSOUND pDirectSound,
  HWND hWnd
);
```

Parameters

ppDirectMusic

Address of a variable that specifies or receives an interface pointer to a DirectMusic object.

If the variable pointed to by *ppDirectMusic* contains a valid **IDirectMusic** or **IDirectMusic8** interface pointer, the existing object is assigned to the performance. The reference count of the interface is incremented. Ports passed to the **IDirectMusicPerformance8::AddPort** method must be created from this DirectMusic object.

If the variable pointed to by *ppDirectMusic* contains NULL, a DirectMusic object is created and an **IDirectMusic** interface pointer is returned. Use **QueryInterface** to obtain **IDirectMusic8**.

If *ppDirectMusic* is NULL, a DirectMusic object is created and used internally by the performance.

See Remarks.

pDirectSound

IDirectSound8 interface pointer to use by default for wave output. If this value is NULL, DirectMusic creates a DirectSound object. There should, however, be only one DirectSound object per process. If your application uses DirectSound separately, it should pass in that interface here, or to **IDirectMusic8::SetDirectSound** if the application creates the DirectMusic object explicitly.

hWnd

Window handle to be used for the creation of DirectSound. This parameter can be NULL, in which case the foreground window is used. See Remarks.

This parameter is ignored if *pDirectSound* is not NULL, in which case the application is responsible for setting the window handle in a call to **IDirectSound8::SetCooperativeLevel**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_ALREADY_INITED

E_OUTOFMEMORY

E_POINTER

Remarks

This method can be called only once. It cannot be used to retrieve an existing **IDirectMusic8** interface.

A DirectMusic object can be associated with the performance in any of the following ways:

- The application creates its own DirectMusic object and gives it to the performance by passing the address of the **IDirectMusic8** pointer in *ppDirectMusic*. In this case, the *pDirectSound* and *hWnd* parameters are ignored because the application is responsible for calling **IDirectMusic8::SetDirectSound**.
- The application allows the performance to create the DirectMusic object and needs a pointer to that object. In this case, **ppDirectMusic* is NULL on entry, and contains the **IDirectMusic** pointer on exit.
- The application allows the performance to initialize itself and does not need a DirectMusic object pointer. In this case, *ppDirectMusic* is NULL.

The performance must be terminated by using the **IDirectMusicPerformance8::CloseDown** method before being released.

You can pass NULL in the *hWnd* parameter to pass the current foreground window handle to DirectSound. However, do not assume that the application window will be in the foreground during initialization. It is best to pass the top-level application window handle.

Requirements

Header: Declared in dmusici.h.

See Also

Migrating from Ports to Audiopaths

IDirectMusicPerformance8::InitAudio

Initializes the performance and optionally sets up a default audiopath. This method must be called before the performance can play using audiopaths.

This method should be used in most cases instead of **IDirectMusicPerformance8::Init**.

```
HRESULT InitAudio(
  IDirectMusic** ppDirectMusic,
  IDirectSound** ppDirectSound,
  HWND hWnd,
  DWORD dwDefaultPathType,
  DWORD dwPChannelCount,
  DWORD dwFlags,
  DMUS_AUDIOPARAMS *pParams
);
```

Parameters

ppDirectMusic

Address of a variable that specifies or receives an interface pointer to a DirectMusic object.

If the variable pointed to by *ppDirectMusic* contains a valid **IDirectMusic** or **IDirectMusic8** interface pointer, the existing object is assigned to the performance. The reference count of the interface is incremented.

If the variable pointed to by *ppDirectMusic* contains NULL, a DirectMusic object is created and the **IDirectMusic** interface pointer is returned. Use **QueryInterface** to obtain **IDirectMusic8**.

If *ppDirectMusic* is NULL, a DirectMusic object is created and used internally by the performance.

See Remarks.

ppDirectSound

Address of a variable that specifies or receives an **IDirectSound** interface pointer for a DirectSound object to use by default for wave output. If this parameter contains a NULL pointer, DirectMusic creates a private DirectSound object. If the variable pointed to contains NULL, DirectMusic creates a DirectSound object and returns the interface pointer. See Remarks.

hWnd

Window handle to use for the creation of DirectSound. This parameter can be NULL, in which case the foreground window is used. See Remarks.

This parameter is ignored if an **IDirectSound** interface pointer is passed to the method in *ppDirectSound*. In that case the application is responsible for setting the window handle by using **IDirectSound8::SetCooperativeLevel**.

dwDefaultPathType

DWORD value that specifies the default audiopath type. Can be zero if no default path type is wanted. For a list of defined values, see **IDirectMusicPerformance8::CreateStandardAudioPath**.

dwPChannelCount

Value that specifies the number of performance channels to allocate to the path, if *dwDefaultPathType* is not zero.

dwFlags

Flags that specify requested features. If *pParams* is not NULL, this value is ignored and the requested features are specified in the **dwFeatures** member of the **DMUS_AUDIOPARAMS** structure. The following values are defined for use in this parameter:

DMUS_AUDIOF_3D
3-D buffers.

DMUS_AUDIOF_ALL
All features.

DMUS_AUDIOF_BUFFERS
Multiple buffers.

DMUS_AUDIOF_ENVIRON
Environmental modeling.

DMUS_AUDIOF_EAX
EAX effects.

DMUS_AUDIOF_STREAMING
Support for streaming waves.

pParams
Address of a **DMUS_AUDIOPARAMS** structure that specifies parameters for the synthesizer and receives information about what parameters were set. Can be NULL if the default parameters are wanted.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

DMUS_E_ALREADY_INITED
DSERR_BUFFERLOST
DSERR_PRIOLEVELNEEDED
DSERR_UNINITIALIZED
E_NOINTERFACE
E_OUTOFMEMORY
E_POINTER

Remarks

This method can be called only once. It cannot be used to retrieve an existing **IDirectMusic8** interface.

A DirectMusic object can be associated with the performance in the following ways.

- The application allows the performance to create the DirectMusic object and needs a pointer to that object. In this case, **ppDirectMusic* is NULL on entry and contains the **IDirectMusic** pointer on exit.
- The application allows the performance to initialize itself and does not need a DirectMusic object pointer. In this case, *ppDirectMusic* is NULL.
- The application creates its own DirectMusic object and gives it to the performance by passing the address of the **IDirectMusic8** pointer in *ppDirectMusic*. Most applications do not use this technique.

If you specify an interface pointer in *ppDirectSound*, it must be an interface to an object of class CLSID_DirectSound8. Objects of this class support both **IDirectSound** and **IDirectSound8**, but the **IDirectSound** interface must be passed. For information on how to create an object of this class, see the Remarks for

IDirectSound8. The DirectSound object must be fully initialized before being passed to **InitAudio**. If the object was created by using **CoCreateInstance**, call **IDirectSound8::Initialize**. Set the cooperative level to DSSCL_PRIORITY by using **IDirectSound8::SetCooperativeLevel**.

You can pass NULL in the *hWnd* parameter to pass the current foreground window handle to DirectSound. However, do not assume that the application window will be in the foreground during initialization. It is best to pass the top-level application window handle.

The parameters set in *dwFlags* and *pParams* apply to the default audiopath and any audiopaths created subsequently.

The method fails with DSERR_BUFFERLOST if a value other than zero is passed in *dwDefaultPathType* and any application has initialized DirectSound with the write-primary cooperative level.

The performance must be terminated by using the **IDirectMusicPerformance8::CloseDown** method before being released.

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::Invalidate

Flushes all queued messages from the supplied time forward and causes all tracks of all segments to resend their data from the given time forward.

```
HRESULT Invalidate(
  MUSIC_TIME mtTime,
  DWORD dwFlags
);
```

Parameters

mtTime

Time from which to invalidate, adjusted by *dwFlags*. Setting this value to 0 causes immediate invalidation.

dwFlags

Flags that adjust *mtTime* to align to measures, beats, or grids. This value can be 0 or one of the following members of the **DMUS_SEGF_FLAGS** enumeration:

DMUS_SEGF_MEASURE
DMUS_SEGF_BEAT
DMUS_SEGF_GRID

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_NO_MASTER_CLOCK.

Remarks

If *mtTime* is so long ago that it is impossible to invalidate that time, the earliest possible time is used.

Requirements

Header: Declared in dmusici.h.

See Also

Prepare Time, Segment Timing

IDirectMusicPerformance8::IsPlaying

Determines whether a particular segment or segment state is currently being heard from the speakers.

```
HRESULT IsPlaying(
  IDirectMusicSegment* pSegment,
  IDirectMusicSegmentState* pSegState
);
```

Parameters

pSegment
Segment to check. If NULL, check only *pSegState*.

pSegState
Segment state to check. If NULL, check only *pSegment*.

Return Values

If the method succeeds and the requested segment or segment state is playing, the return value is S_OK. If neither is playing or only one was requested and it is not playing, the return value is S_FALSE.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_NO_MASTER_CLOCK

Remarks

The method returns S_OK only if the segment or segment state is actually playing at the speakers. Because of latency, this method might return S_FALSE even though **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx** has just been called on the segment. Similarly, the method returns S_OK as long as the segment is being heard, even though all messages might already have been dispatched.

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::MIDIToMusic

Converts a MIDI note value to a DirectMusic music value, using a supplied chord, subchord level, and play mode.

```
HRESULT MIDIToMusic(
  BYTE bMIDIValue,
  DMUS_CHORD_KEY* pChord,
  BYTE bPlayMode,
  BYTE bChordLevel,
  WORD *pwMusicValue
);
```

Parameters

bMIDIValue
MIDI note value to convert, in the range from 0 through 127.

pChord
Address of a **DMUS_CHORD_KEY** structure containing information about the chord and key structure to be used in translating the note. This includes the underlying scale. For example, if the chord is a CM7, the note is interpreted against the chord positions for root note C, chord intervals of a major seventh. The structure carries up to DMUS_MAXSUBCHORD parallel subchords, with chord intervals, root, scale, and inversion flags for each. It also carries the overall key root.

bPlayMode
Play mode determining how the music value is derived from the chord. For a list of values, see **DMUS_PLAYMODE_FLAGS**.

bChordLevel
Subchord level, defining which subchords can be used. See **DMUS_SUBCHORD**.

pwMusicValue

Address of a variable that receives the music value. For information on this value, see **DMUS_NOTE_PMSG**.

Return Values

If the method succeeds, the return value is one of the following. See Remarks.

S_OK
DMUS_S_DOWN_OCTAVE
DMUS_S_UP_OCTAVE

If it fails, the method can return one of the following error values:

DMUS_E_CANNOT_CONVERT
E_INVALIDARG

Remarks

If the method fails, **pwMusicValue* is not changed.

If the return value is DMUS_S_UP_OCTAVE or DMUS_DOWN_OCTAVE, the note conversion generated a note value that is less than 0 or greater than 127, so it has been bumped up or down one or more octaves to be in the proper MIDI range of from 0 through 127. This can occur when using play modes DMUS_PLAYMODE_FIXEDTOCHORD and DMUS_PLAYMODE_FIXEDTOKEY, both of which return MIDI values in **pwMusicValue*.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::MusicToMIDI, Music Values and MIDI Notes

IDirectMusicPerformance8::MusicToMIDI

Converts a DirectMusic music value to a MIDI note value.

```
HRESULT MusicToMIDI(
  WORD wMusicValue,
  DMUS_CHORD_KEY* pChord,
  BYTE bPlayMode,
  BYTE bChordLevel,
  BYTE *pbMIDIValue
);
```

Parameters

wMusicValue
Music value to convert. For information on music values, see **DMUS_NOTE_PMSG**.

pChord
Address of a **DMUS_CHORD_KEY** structure containing information about the chord and key structure to be used in translating the note. This includes the underlying scale. For example, if the chord is a CM7, the note is interpreted against the chord positions for root note C, chord intervals of a major seventh. The structure carries up to DMUS_MAXSUBCHORD parallel subchords, with chord intervals, root, scale, and inversion flags for each. It also carries the overall key root.

bPlayMode
Play mode determining how the music value is related to the chord. For a list of values, see **DMUS_PLAYMODE_FLAGS**.

bChordLevel
Subchord level, defining which subchords can be used. See **DMUS_SUBCHORD**.

pbMIDIValue
Address of a variable that receives the MIDI value, in the range from 0 through 127.

Return Values

If the method succeeds, the return value is one of the following. See Remarks.

S_OK
DMUS_S_OVER_CHORD
DMUS_S_DOWN_OCTAVE
DMUS_S_UP_OCTAVE

If it fails, the method can return one of the following error values:

DMUS_E_CANNOT_CONVERT
E_INVALIDARG

Remarks

If the method fails or returns DMUS_S_OVER_CHORD, **pwMIDIValue* is not changed.

The method returns DMUS_S_OVER_CHORD if no note has been calculated because the music value has the note at a position higher than the top note of the chord. This applies only to DMUS_PLAYMODE_NORMALCHORD play mode.

The caller should not do anything with the note, which is not meant to be played against this chord.

If the return value is DMUS_S_UP_OCTAVE or DMUS_DOWN_OCTAVE, the note conversion generated a note value that is less than 0 or greater than 127, so it has been bumped up or down one or more octaves to be in the proper MIDI range of 0 through 127. This can occur when using any play mode except DMUS_PLAYMODE_FIXED.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::MIDIToMusic, Music Values and MIDI Notes

IDirectMusicPerformance8::MusicToReferenceTime

Converts a performance time in **MUSIC_TIME** format to performance time in **REFERENCE_TIME** format.

```
HRESULT MusicToReferenceTime(
  MUSIC_TIME mtTime,
  REFERENCE_TIME* prtTime
);
```

Parameters

mtTime
Time in **MUSIC_TIME** format to convert.

prtTime
Address of a variable that receives the converted time in **REFERENCE_TIME** format.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_NO_MASTER_CLOCK

Remarks

Because reference time has a greater precision than music time, a time that has been converted from reference time to music time, and then back again, probably does not have its original value.

This method converts a time offset from the start of the performance, not a duration. Because the ratio between music time and reference time units depends on the tempo, DirectMusic takes into account all tempo changes since the start of the performance when calculating *prtTime*. If a master tempo has been set for the performance, it is taken into account as well.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::ReferenceToMusicTime, Clock Time vs. Music Time

IDirectMusicPerformance8::PChannelInfo

Retrieves the port, group, and MIDI channel for a given performance channel.

```
HRESULT PChannelInfo(
  DWORD dwPChannel,
  IDirectMusicPort** ppPort,
  DWORD* pdwGroup,
  DWORD* pdwMChannel
);
```

Parameters

dwPChannel
Performance channel for which information is desired.

ppPort
Address of a variable that receives an **IDirectMusicPort8** pointer. This value can be NULL if the pointer is not wanted. If a non-NULL pointer is returned, the reference count is incremented, and it is the responsibility of the application to call **Release** on the pointer. See also Remarks.

pdwGroup
Address of a variable that receives the group on the port. Can be NULL if this value is not wanted.

pdwMChannel

Address of a variable that receives the MIDI channel on the group. Can be NULL if this value is not wanted.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_POINTER

Remarks

A NULL pointer is returned in **ppPort* if the port has been removed by a call to **IDirectMusicPerformance8::RemovePort**, but the method succeeds.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::AssignPChannel,
IDirectMusicPerformance8::AssignPChannelBlock

IDirectMusicPerformance8::PlaySegment

Begins playback of a segment.

```
HRESULT PlaySegment(
  IDirectMusicSegment* pSegment,
  DWORD dwFlags,
  __int64 i64StartTime,
  IDirectMusicSegmentState** ppSegmentState
);
```

Parameters

pSegment
Segment to play.

dwFlags
Flags that modify the method's behavior. See **DMUS_SEGF_FLAGS**.

i64StartTime
Performance time at which to begin playing the segment, adjusted to any resolution boundary specified in *dwFlags*. The time is in music time unless the

DMUS_SEGF_REFTIME flag is set. A value of 0 causes the segment to start playing as soon as possible.

ppSegmentState

Address of a variable that receives a pointer to the segment state for this instance of the playing segment. This field can be NULL. If it is non-NULL, the segment state pointer is returned, and the application must call **Release** on it.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY

E_POINTER

DMUS_E_NO_MASTER_CLOCK

DMUS_E_SEGMENT_INIT_FAILED

DMUS_E_TIME_PAST

Remarks

Segments should be greater than 250 milliseconds in length.

The boundary resolutions in *dwFlags* are relative to the primary segment.

If a primary segment is scheduled to play while another primary segment is playing, the first one stops unless you set the DMUS_SEGF_QUEUE flag for the second segment, in which case it plays as soon as the first one finishes.

For more information on the exact start time of segments, see Segment Timing. For information on how the start time of segments can be affected by tempo changes, see Clock Time vs. Music Time.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::PlaySegmentEx

IDirectMusicPerformance8::PlaySegmentEx

Begins playback of a segment. The method offers greater functionality than **IDirectMusicPerformance8::PlaySegment**.

HRESULT PlaySegmentEx(

```
  IUnknown* pSource,
  WCHAR *pwzSegmentName,
  IUnknown* pTransition,
  DWORD dwFlags,
  __int64 i64StartTime,
  IDirectMusicSegmentState** ppSegmentState,
  IUnknown* pFrom,
  IUnknown* pAudioPath
);
```

Parameters

pSource
Address of the **IUnknown** interface of the object to play.

pwzSegmentName
Not implemented in DirectX 8.0. Set to NULL.

pTransition
IUnknown interface pointer of a template segment to use in composing a transition to this segment. Can be NULL. See Remarks.

dwFlags
Flags that modify the method's behavior. See **DMUS_SEGF_FLAGS**.

i64StartTime
Performance time at which to begin playing the segment, adjusted to any resolution boundary specified in *dwFlags*. The time is in music time unless the DMUS_SEGF_REFTIME flag is set. A value of zero causes the segment to start playing as soon as possible.

ppSegmentState
Address of a variable that receives an **IDirectMusicSegmentState** interface pointer for this instance of the playing segment. Use **QueryInterface** to obtain **IDirectMusicSegmentState8**. The reference count of the interface is incremented. This parameter can be NULL if no segment state pointer is wanted.

pFrom
IUnknown interface pointer of a segment state or audiopath to stop when the new segment begins playing. If it is an audiopath, all segment states playing on that audiopath are stopped. This value can be NULL.

pAudioPath
IUnknown interface pointer of an object that represents the audiopath on which to play, or NULL to play on the default path.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

DMUS_E_AUDIOPATH_INACTIVE
DMUS_E_AUDIOPATH_NOPORT
DMUS_E_NO_MASTER_CLOCK
DMUS_E_SEGMENT_INIT_FAILED
DMUS_E_TIME_PAST
E_OUTOFMEMORY
E_POINTER

Remarks

Segments should be greater than 250 milliseconds in length.

The boundary resolutions in *dwFlags* are relative to the primary segment.

If a primary segment is scheduled to play while another primary segment is playing, the first one stops unless you set the DMUS_SEGF_QUEUE flag for the second segment, in which case it plays as soon as the first one finishes.

For more information on the exact start time of segments, see Segment Timing. For information on how the start time of segments can be affected by tempo changes, see Clock Time vs. Music Time.

If DMUS_SEGF_AUTOTRANSITION is specified in *dwFlags* and a segment is already playing at *i64StartTime* and is being interrupted, the method composes a transition between the two segments and plays it before playing *pSource*. The transition is based a template provided at *pTransition*.

The method can be used to play on a performance that does not use audiopaths; that is, one initialized by using **IDirectMusicPerformance8::Init**. In this case the *pAudioPath* parameter must be NULL.

Requirements

Header: Declared in dmusici.h.

IDirectMusicPerformance8::ReferenceToMusicTime

Converts a performance time in **REFERENCE_TIME** format to a performance time in **MUSIC_TIME** format.

```
HRESULT ReferenceToMusicTime(
  REFERENCE_TIME rtTime,
  MUSIC_TIME* pmtTime
);
```

Parameters

rtTime
Time in **REFERENCE_TIME** format.

pmtTime
Address of a variable that receives the converted time in **MUSIC_TIME** format.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

E_POINTER
DMUS_E_NO_MASTER_CLOCK

Remarks

Because music time is less precise than reference time, rounding occurs.

This method converts a time offset from the start of the performance, not a duration. Because the ratio between music time and reference time units depends on the tempo, DirectMusic takes into account all tempo changes since the start of the performance when calculating *prtTime*.If a master tempo has been set for the performance, it is taken into account as well.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::MusicToReferenceTime, Clock Time vs. Music Time

IDirectMusicPerformance8::RemoveNotificationType

Removes a previously added notification type from the performance. All segments and tracks are updated by a call to their **RemoveNotificationType** methods.

```
HRESULT RemoveNotificationType(
  REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType

Reference to (C++) or address of (C) the identifier of the notification type to remove. (For the defined types, see **DMUS_NOTIFICATION_PMSG**.) If this value is GUID_NULL, all notifications are to be removed.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE (see Remarks).

If it fails, the method can return E_POINTER.

Remarks

S_FALSE is returned when *rguidNotificationType* is not an active notification.

If a notification was added to a segment that has stopped playing, the performance cannot remove the notification type from that segment because it no longer has a reference to the segment.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::AddNotificationType,
IDirectMusicSegment8::RemoveNotificationType,
IDirectMusicTrack8::RemoveNotificationType, Notification and Event Handling

IDirectMusicPerformance8::RemovePort

Removes a port from the performance. Any performance channels that map to this port are invalidated, and messages stamped with them do not play.

```
HRESULT RemovePort(
  IDirectMusicPort* pPort
);
```

Parameters

pPort
Port to remove.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG

E_POINTER

Remarks

A port added by passing NULL to **IDirectMusicPerformance8::AddPort** cannot be removed by passing NULL to **RemovePort**.

This method should not be called by applications that use audiopaths.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::AddPort

IDirectMusicPerformance8::RhythmToTime

Converts rhythm time to music time.

```
HRESULT RhythmToTime(
  WORD wMeasure,
  BYTE bBeat,
  BYTE bGrid,
  short nOffset,
  DMUS_TIMESIGNATURE *pTimeSig,
  MUSIC_TIME *pmtTime
);
```

Parameters

wMeasure
Measure of the time to convert.

bBeat
Beat of the time to convert.

bGrid
Grid of the time to convert.

nOffset
Offset from the grid, in music-time ticks, of the time to convert.

pTimeSig
Address of a **DMUS_TIMESIGNATURE** structure containing information about the time signature.

pmtTime
Address of a variable that receives the music time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::TimeToRhythm

IDirectMusicPerformance8::SendPMsg

Sends a performance message. This method is called by tracks when they are played. It might also be called by a tool to inject new data into a performance.

```
HRESULT SendPMsg(
  DMUS_PMSG* pPMSG
);
```

Parameters

pPMSG
Message allocated by **IDirectMusicPerformance8::AllocPMsg**. This structure is of a type derived from **DMUS_PMSG**. See DirectMusic Messages.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
E_INVALIDARG
E_POINTER

Remarks

The **dwFlags** member (see **DMUS_PMSG**) must contain either DMUS_PMSGF_MUSICTIME or DMUS_PMSGF_REFTIME, depending on the time stamp in either **rtTime** or **mtTime**. The **dwFlags** member should also contain

the appropriate delivery type—DMUS_PMSGF_TOOL_QUEUE, DMUS_PMSGF_TOOL_ATTIME, or DMUS_PMSGF_TOOL_IMMEDIATE—depending on the type of message. If none is selected, DMUS_PMSGF_TOOL_IMMEDIATE is used by default.

If the time of the message is set to 0 and the **dwFlags** member contains DMUS_PMSGF_REFTIME, it is assumed that this message is cued to go out immediately.

In most cases, the **IDirectMusicGraph8::StampPMsg** method should be called on the message before **SendPMsg** is called. However, when sending a message directly to the main output tool, this step can be skipped. If you want the message to pass only through the performance graph, obtain the **IDirectMusicGraph8** interface by calling **IDirectMusicPerformance8::QueryInterface**. Otherwise, obtain it by calling **IDirectMusicSegment8::QueryInterface**. Do not attempt to obtain the interface by calling **IDirectMusicPerformance8::GetGraph** or **IDirectMusicSegment8::GetGraph**; these methods return a pointer to the graph object, rather than to the implementation of the **IDirectMusicGraph8** interface on the performance or segment.

Normally, the performance frees the message after it has been processed. For more information, see the Remarks for **IDirectMusicPerformance8::FreePMsg**.

The follow code example shows how to allocate and send a system exclusive message and a tempo message.

```
// Assume that pPerformance is a valid IDirectMusicPerformance8
// pointer and that mtTime is an initialized MUSIC_TIME
// variable.

IDirectMusicGraph* pGraph;

// Get the graph pointer from the performance. If you wanted the
// message to go through a segment graph, you would
// QueryInterface a segment object instead.

if ( SUCCEEDED( pPerformance->QueryInterface(
      IID_IDirectMusicGraph, (void**)&pGraph )))
{
   // Allocate a DMUS_SYSEX_PMSG of the appropriate size,
   // and read the system exclusive data into it.

   DMUS_SYSEX_PMSG* pSysEx;

   if ( SUCCEEDED( pPerformance->AllocPMsg(
         sizeof(DMUS_SYSEX_PMSG) + m_dwSysExLength,
         (DMUS_PMSG**)&pSysEx )))
   {
      ZeroMemory(pSysEx, sizeof(DMUS_NOTE_PMSG));
```

```
    pSysEx->dwSize = sizeof(DMUS_SYSEX_PMSG);
    pSysEx->dwLen = dwSysExLength;
    pSysEx->mtTime = mtTime;
    pSysEx->dwFlags = DMUS_PMSGF_MUSICTIME;
    pSysEx->dwType = DMUS_PMSGT_SYSEX;

    // Assume that m_pbSysExData is a pointer to an array
    // containing data of length m_dwSysExLength.

    memcpy( pSysEx->abData, m_pbSysExData, m_dwSysExLength );

    pGraph->StampPMsg( (DMUS_PMSG*)pSysEx );
    if (FAILED(pPerformance->SendPMsg( (DMUS_PMSG*)pSysEx )))
    {
        pPerformance->FreePMsg( (DMUS_PMSG*)pSysEx );
    }
}

// Change the tempo at time mtTime to 120 bpm.

DMUS_TEMPO_PMSG* pTempo;

if( SUCCEEDED( pPerformance->AllocPMsg(
        sizeof(DMUS_TEMPO_PMSG),
        (DMUS_PMSG**)&pTempo )))
{
    pTempo->dwSize = sizeof(DMUS_TEMPO_PMSG);
    pTempo->dblTempo = 120;
    pTempo->mtTime = mtTime;
    pTempo->dwFlags = DMUS_PMSGF_MUSICTIME;
    pTempo->dwType = DMUS_PMSGT_TEMPO;
    pGraph->StampPMsg( (DMUS_PMSG*)pTempo );
    if (FAILED(pPerformance->SendPMsg( (DMUS_PMSG*)pTempo )))
    {
        pPerformance->FreePMsg( (DMUS_PMSG*)pTempo );
    }
}

pGraph->Release();
}
```

The next code example shows a function that sends a note message associated with the track identified by *dwTrackID*. The virtual track ID should be 0 if the message is not being generated from a DirectMusicTrack object.

```
HRESULT CreateNotePMsg(IDirectMusicPerformance8* pPerformance,
```

```
    MUSIC_TIME mtTime, DWORD dwTrackID)
{

  // Allocate a Note PMessage.
  DMUS_NOTE_PMSG* pNote = NULL;
  HRESULT hr = pPerformance->AllocPMsg( sizeof(DMUS_NOTE_PMSG),
      (DMUS_PMSG**) &pNote);
  if (FAILED(hr)) return hr;

  pNote->dwSize = sizeof(DMUS_NOTE_PMSG); // Size of a Note Pmessage.
  pNote->rtTime = 0;                  // Ignored.
  pNote->mtTime = mtTime;             // When to play the note.
  pNote->dwFlags = DMUS_PMSGF_MUSICTIME;  // Use the mtTime field.
  pNote->dwPChannel = 5;              // Play on PChannel 5.
  pNote->dwVirtualTrackID = dwTrackID;    // Track ID from parameter.

  // The following two fields should be set to NULL when a
  // message is initially sent. They will be updated in
  // IDirectMusicGraph::StampPMsg.
  pNote->pTool = NULL;
  pNote->pGraph = NULL;
  pNote->dwType = DMUS_PMSGT_NOTE;
  pNote->dwVoiceID = 0;          // Always 0
  pNote->dwGroupID = 0xFFFFFFFF;  // All track groups
  pNote->punkUser = NULL;        // Always NULL

  // Get the current time signature from the performance
  // to compute measure and beat information.
  DMUS_TIMESIGNATURE TimeSig;
  MUSIC_TIME mtNext;
  hr = pPerformance->GetParam(GUID_TimeSignature, 0xFFFFFFFF,
      0, mtTime, &mtNext, &TimeSig);
  if (FAILED(hr)) return hr;

  // Recompute TimeSig.mtTime to have the value expected
  // by pPerformance->TimeToRhythm.
  TimeSig.mtTime += mtTime;

  // Get the current chord from the performance
  // to create a note value.
  DMUS_CHORD_KEY Chord;
  hr = pPerformance->GetParam(GUID_ChordParam, 0xFFFFFFFF, 0,
      mtTime, &mtNext, &Chord);
  if (FAILED(hr)) return hr;

  // Create a note with octave 5, chord tone 2 (fifth), scale
```

```
// offset 1 (=> sixth), and no accidentals.
WORD wMusicValue = 0x5210;

// Use DMUS_PLAYMODE_PEDALPOINT as your play mode
// in pPerformance->MusicToMIDI.
BYTE bPlayModeFlags = DMUS_PLAYMODE_PEDALPOINT;

// Fill in the fields specific to DMUS_NOTE_PMSG.
pNote->wMusicValue = wMusicValue;
hr = pPerformance->MusicToMIDI(
   wMusicValue,
   &Chord,
   bPlayModeFlags,
   0,
   &(pNote->bMidiValue));
if (FAILED(hr)) return hr;

hr = pPerformance->TimeToRhythm(
   TimeSig.mtTime,
   &TimeSig,
   &(pNote->wMeasure),
   &(pNote->bBeat),
   &(pNote->bGrid),
   &(pNote->nOffset));
if (FAILED(hr)) return hr;

pNote->mtDuration = DMUS_PPQ;       // Quarter note duration.
pNote->bVelocity = 120;             // MIDI velocity (0 to 127).
pNote->bFlags = DMUS_NOTEF_NOTEON;  // Always set to this value.
pNote->bTimeRange = 250;            // Randomize start time a lot.
pNote->bDurRange = 5;               // Randomize duration a little.
pNote->bVelRange = 0;               // Don't randomize velocity.
pNote->bPlayModeFlags = bPlayModeFlags;
pNote->bSubChordLevel = 0;          // Note uses subchord level 0.
pNote->cTranspose = 0;              // No transposition.

// Stamp the message with the performance graph.
IDirectMusicGraph* pGraph;
hr = pPerformance->QueryInterface( IID_IDirectMusicGraph,
      (void**)&pGraph );
if (FAILED(hr)) return hr;

pGraph->StampPMsg( (DMUS_PMSG*)pNote );
pGraph->Release();

// Finally, send the message.
```

```
    hr = pPerformance->SendPMsg( (DMUS_PMSG*)pNote);
    if (FAILED(hr))
    {
        pPerformance->FreePMsg( (DMUS_PMSG*)pNote);
        return hr;
    }
    return S_OK;
}
```

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicTool8::ProcessPMsg, Messages, DirectMusic Messages, DirectMusic Tools

IDirectMusicPerformance8::SetBumperLength

Sets the amount of time to buffer ahead of the port's latency for messages to be sent to the port for rendering. For an overview of this topic, see Timing.

```
HRESULT SetBumperLength(
  DWORD dwMilliSeconds
);
```

Parameters

dwMilliSeconds
Amount of preplay time, in milliseconds. The default value is 50.

Return Values

The method returns S_OK.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetBumperLength, **IDirectMusicPerformance8::SetPrepareTime**

IDirectMusicPerformance8::SetDefaultAudioPath

Sets and activates the default audiopath for the performance.

```
HRESULT SetDefaultAudioPath(
  IDirectMusicAudioPath *pAudioPath
);
```

Parameters

pAudioPath
Pointer to the **IDirectMusicAudioPath8** interface of the default audiopath, or NULL to remove the current default audiopath.

Return Values

If it succeeds, the method returns S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_AUDIOPATH_NOPORT
DMUS_E_AUDIOPATHS_NOT_VALID
DMUS_E_NOT_INIT
E_INVALIDARG
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetDefaultAudioPath,
IDirectMusicPerformance8::InitAudio.

IDirectMusicPerformance8::SetGlobalParam

Sets global values for the performance.

```
HRESULT SetGlobalParam(
  REFGUID rguidType,
  void* pParam,
  DWORD dwSize
```

);

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data.

pParam
Address of data to be copied and stored by the performance.

dwSize
Size of the data. This is constant for each *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_POINTER
E_OUTOFMEMORY

Remarks

The *dwSize* parameter is needed because the performance does not know about all types of data. New types can be created as needed.

For the parameters defined by DirectMusic and their associated data types, see Setting and Retrieving Global Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetGlobalParam, **IDirectMusicPerformance8::SetParam**, Performance Parameters

IDirectMusicPerformance8::SetGraph

Replaces the performance's toolgraph.

```
HRESULT SetGraph(
  IDirectMusicGraph* pGraph
);
```

Parameters

pGraph
Toolgraph to set. Can be set to NULL to clear the graph from the performance.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

Any messages flowing through tools in the current toolgraph are deleted.

Because the graph's reference count is incremented by this method, it is safe to release the original reference.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetGraph, **IDirectMusicPerformance8::GetGraph**, **IDirectMusicPerformance8::SendPMsg**

IDirectMusicPerformance8::SetNotificationHandle

Sets the event handle (created by the Microsoft® Win32® **CreateEvent** function) for notifications. The application should use the Win32 **WaitForSingleObject** function on this handle. When signaled, the application should call the **IDirectMusicPerformance8::GetNotificationPMsg** method to retrieve the notification event.

```
HRESULT SetNotificationHandle(
  HANDLE hNotification,
  REFERENCE_TIME rtMinimum
);
```

Parameters

hNotification
Event handle created by **CreateEvent**, or 0 to clear out an existing handle.

rtMinimum

Minimum time that the performance should hold onto old notify events before discarding them. The value 0 means to use the default minimum time of 20,000,000 reference time units, which is 2 seconds, or the previous value if this method has been called previously. If the application has not called **GetNotificationPMsg** by this time, the event is discarded to free the memory.

Return Values

The method returns S_OK.

Remarks

It is the application's responsibility to call the Win32 **CloseHandle** function on the notification handle when it is no longer needed.

Requirements

Header: Declared in dmusici.h.

See Also

Notification and Event Handling

IDirectMusicPerformance8::SetParam

Sets data on a track inside the control segment.

```
HRESULT SetParam(
  REFGUID rguidType,
  DWORD dwGroupBits,
  DWORD dwIndex,
  MUSIC_TIME mtTime,
  void* pParam
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data to set. See Standard Track Parameters.

dwGroupBits
Group that the desired track is in.

dwIndex
Index of the track in the group identified by *dwGroupBits* in which data is to be set, or DMUS_SEG_ALLTRACKS to set the parameter on all tracks in the group that contain the parameter.

mtTime
Time at which to set the data. Unlike **IDirectMusicSegment8::SetParam**, this time is in performance time. The start time of the segment is subtracted from this time, and the result is passed to **IDirectMusicSegment8::SetParam**.

pParam
Address of a structure containing the data. This structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND
E_POINTER

Remarks

Normally the primary segment is the control segment. However, a secondary segment can be designated as the control segment when it is played. See **DMUS_SEGF_FLAGS**.

For an explanation of *dwGroupBits* and *dwIndex*, see Identifying the Track.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetGlobalParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::SetParamEx**, **IDirectMusicPerformance8::GetTime**, Performance Parameters

IDirectMusicPerformance8::SetPrepareTime

Sets the interval between the time when messages are sent by tracks and the time when the sound is heard. This interval allows sufficient time for the message to be processed by tools.

HRESULT SetPrepareTime(
DWORD *dwMilliSeconds*

```
);
```

Parameters

dwMilliSeconds
Amount of prepare time, in milliseconds. The default value is 1000.

Return Values

The method returns S_OK.

Remarks

For an overview, see Timing.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetPrepareTime, **IDirectMusicPerformance8::SetBumperLength**, Prepare Time, Latency and Bumper Time

IDirectMusicPerformance8::Stop

Stops playback of a segment or segment state.

This method has been superseded by **IDirectMusicPerformance8::StopEx**, which can stop playback of a segment, segment state, or audiopath.

```
HRESULT Stop(
  IDirectMusicSegment* pSegment,
  IDirectMusicSegmentState* pSegmentState,
  MUSIC_TIME mtTime,
  DWORD dwFlags
);
```

Parameters

pSegment
Segment to stop playing. All segment states based on this segment are stopped at *mtTime*. See Remarks.

pSegmentState
Segment state to stop playing. See Remarks.

mtTime
Time at which to stop the segment, segment state, or both. If the time is in the past or if 0 is passed in this parameter, the specified segment and segment states stop playing immediately.

dwFlags
Flag that indicates when the stop should occur. Boundaries are in relation to the current primary segment. For a list of values, see **IDirectMusicPerformance8::StopEx**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

If *pSegment* and *pSegmentState* are both NULL, all music stops, and all currently cued segments are released. If either *pSegment* or *pSegmentState* is not NULL, only the requested segment states are removed from the performance. If both are non-NULL and DMUS_SEGF_DEFAULT is used, the default resolution from the *pSegment* is used.

If you set all parameters to NULL or 0, everything stops immediately, and controller reset messages and note-off messages are sent to all mapped performance channels.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::PlaySegment,
IDirectMusicPerformance8::PlaySegmentEx,
IDirectMusicPerformance8::StopEx, **DMUS_SEGF_FLAGS**

IDirectMusicPerformance8::StopEx

Stops playback of an object.

```
HRESULT StopEx(
  IUnknown *pObjectToStop,
  __int64 i64StopTime,
  DWORD dwFlags
);
```

Parameters

pObjectToStop
Pointer to the **IUnknown** interface of the segment, segment state, or audiopath to stop.

i64StopTime
Time at which to stop. If the time is in the past or if 0 is passed in this parameter, the object stops playing immediately.

dwFlags
Flags that indicate when the stop should occur. Boundaries are in relation to the current primary segment. Can be one of the following values, or DMUS_SEGF_REFTIME combined with one other.

0
Stop immediately.

DMUS_SEGF_AUTOTRANSITION
Compose and play a transition to silence. Valid only when stopping a song. Not implemented in DirectX 8.0.

DMUS_SEGF_BEAT
Stop on the next beat boundary at or after *i64StopTime*.

DMUS_SEGF_DEFAULT
Stop on the default boundary, as set by the **IDirectMusicSegment8::SetDefaultResolution** method.

DMUS_SEGF_GRID
Stop on the next grid boundary at or after *i64StopTime*.

DMUS_SEGF_MEASURE
Stop on the next measure boundary at or after *i64StopTime*.

DMUS_SEGF_REFTIME
The value in *i64StopTime* is in reference time.

DMUS_SEGF_SEGMENTEND
Stop at the end of the primary segment.

DMUS_SEGF_MARKER
Stop at the next marker.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

Stopping a segment stops all instances that are playing.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::Stop, **IDirectMusicPerformance8::PlaySegmentEx**, **DMUS_SEGF_FLAGS**

IDirectMusicPerformance8::TimeToRhythm

Converts music time to rhythm time.

```
HRESULT TimeToRhythm(
  MUSIC_TIME mtTime,
  DMUS_TIMESIGNATURE *pTimeSig,
  WORD *pwMeasure,
  BYTE *pbBeat,
  BYTE *pbGrid,
  short *pnOffset
);
```

Parameters

mtTime
Time to convert.

pTimeSig
Address of a **DMUS_TIMESIGNATURE** structure that contains information about the time signature.

pwMeasure
Address of a variable that receives the measure in which the time falls.

pbBeat
Address of a variable that receives the beat at which the time falls.

pbGrid
Address of a variable that receives the grid at which the time falls.

pnOffset
Address of a variable that receives the offset from the grid (in music-time ticks) at which the time falls.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::RhythmToTime

IDirectMusicPort8

The **IDirectMusicPort8** interface provides access to a DirectMusicPort object, which represents a device that sends or receives music data. The input port of an MPU-401, the output port of an MPU-401, the Microsoft software synthesizer, and an IHV-provided filter are all ports. A physical device such as an MPU-401 might provide multiple ports. A single port, however, cannot both capture and render data.

IDirectMusicPort8 is a define for **IDirectMusicPort**. The two interface names are interchangeable.

The interface is typically obtained by using the **IDirectMusic8::CreatePort** method.

For an overview, see Using DirectMusic Ports.

The methods of the **IDirectMusicPort8** interface can be organized into the following groups.

Buffers	**PlayBuffer**
	Read
	SetReadNotificationHandle
Channels	**GetChannelPriority**
	GetNumChannelGroups
	SetChannelPriority
	SetNumChannelGroups
Device management	**Activate**
	DeviceIoControl
	SetDirectSound
DLS data	**Compact**
	DownloadInstrument
	UnloadInstrument
Information	**GetCaps**
	GetFormat
	GetLatencyClock
	GetRunningStats

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTMUSICPORT8** type is defined as a pointer to the **IDirectMusicPort8** interface.

```
typedef IDirectMusicPort8 *LPDIRECTMUSICPORT8;
```

Requirements

Header: Declared in dmusicc.h.

IDirectMusicPort8::Activate

Activates or deactivates the port.

```
HRESULT Activate(
  BOOL fActive
);
```

Parameters

fActive
Switch to activate (TRUE) or deactivate (FALSE) the port.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DSERR_NODRIVER, indicating that no sound driver is present.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusic8::Activate, Using DirectMusic Ports

IDirectMusicPort8::Compact

Instructs the port to compact DLS or wave-table memory, thus making the largest possible contiguous chunk of memory available for new instruments to be downloaded.

HRESULT Compact();

Parameters

None.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG
E_NOTIMPL
E_OUTOFMEMORY

Remarks

This method affects only DLS devices that need to manage their own DLS wavetable memory. On ports that do not manage their own memory (such as software synthesizers or hardware synthesizers that use host system memory), the method returns E_NOTIMPL.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicPort8::DeviceIoControl

Calls the Win32 **DeviceIoControl** function on the underlying file handle implementing the port.

HRESULT DeviceIoControl(
DWORD *dwIoControlCode*,
LPVOID *lpInBuffer*,
DWORD *nInBufferSize*,
LPVOID *lpOutBuffer*,
DWORD *nOutBufferSize*,
LPDWORD *lpBytesReturned*,
LPOVERLAPPED *lpOverlapped*
);

Parameters

dwIoControlCode

Control code of the operation to perform.

lpInBuffer
Buffer that contains input data.

nInBufferSize
Size of input buffer.

lpOutBuffer
Buffer that receives output data.

nOutBufferSize
Size of the output buffer.

lpBytesReturned
Address of a variable that receives the output byte count.

lpOverlapped
Address of an overlapped structure for asynchronous operation.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_NOTIMPL

Remarks

This method is supported only on ports implemented by a Windows Driver Model (WDM) filter graph. In the case of a WDM filter graph, the file handle used is the topmost pin in the graph.

DirectMusic can refuse to perform defined kernel streaming operations on a pin that might collide with operations that it is performing on the filter graph. User-defined operations, however, are never blocked.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicPort8::DownloadInstrument

Downloads an instrument to the DLS device. Downloading an instrument means handing the data that makes up the instrument to the DLS device. This includes articulation data and all waves needed by the instrument. To save wave space, only waves and articulation required for a range are downloaded. The method returns an **IDirectMusicDownloadedInstrument8** interface pointer, which is later used to unload the instrument.

HRESULT DownloadInstrument(

```
  IDirectMusicInstrument *pInstrument,
  IDirectMusicDownloadedInstrument **ppDownloadedInstrument,
  DMUS_NOTERANGE *pNoteRanges,
  DWORD dwNumNoteRanges;
);
```

Parameters

pInstrument
Pointer to the **IDirectMusicInstrument8** interface of the instrument whose data is downloaded.

ppDownloadedInstrument
Address of a variable that receives a pointer to the **IDirectMusicDownloadedInstrument8** interface.

pNoteRanges
Address of an array of **DMUS_NOTERANGE** structures. Each entry in the array specifies a contiguous range of MIDI note messages to which the instrument must respond. An instrument region is downloaded only if at least one note in that region is specified in the **DMUS_NOTERANGE** structures.

dwNumNoteRanges
Number of **DMUS_NOTERANGE** structures in the array pointed to by *pNoteRanges*. If this value is set to 0, the *pNoteRanges* parameter is ignored, and all regions and wave data for the instrument are downloaded.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_OUTOFMEMORY
E_NOTIMPL

Remarks

To prevent memory loss, the instrument must be unloaded by calling both **IDirectMusicPort8::UnloadInstrument** and **IDirectMusicDownloadedInstrument8::Release** when it is no longer needed.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort8::Compact, Working with Instruments

IDirectMusicPort8::GetCaps

Retrieves the port's capabilities.

```
HRESULT GetCaps(
  LPDMUS_PORTCAPS pPortCaps
);
```

Parameters

pPortCaps
Address of a **DMUS_PORTCAPS** structure that receives the capabilities of the port. The **dwSize** member of this structure must be properly initialized before the method is called.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_INVALIDARG

Requirements

Header: Declared in dmusicc.h.

IDirectMusicPort8::GetChannelPriority

Retrieves the priority of a MIDI channel. For an overview, see Channels.

```
HRESULT GetChannelPriority(
  DWORD   dwChannelGroup,
  DWORD   dwChannel,
  LPDWORD pdwPriority
);
```

Parameters

dwChannelGroup
Group that the channel is in.

dwChannel

Index of the channel on the group.

pdwPriority

Address of a variable that receives the priority ranking. See Remarks.

Return Values

The return value is S_OK.

Remarks

The following values, defined in Dmusicc.h, each represent a range of priorities. They are listed here in descending order of priority.

DAUD_CRITICAL_VOICE_PRIORITY
DAUD_HIGH_VOICE_PRIORITY
DAUD_STANDARD_VOICE_PRIORITY
DAUD_LOW_VOICE_PRIORITY
DAUD_PERSIST_VOICE_PRIORITY

The following values express the default ranking of the channels within a range, according to the DLS Level 1 standard. They are listed here in descending order. Channel 10, the percussion channel, has the highest priority.

DAUD_CHAN10_VOICE_PRIORITY_OFFSET
DAUD_CHAN1_VOICE_PRIORITY_OFFSET
DAUD_CHAN2_VOICE_PRIORITY_OFFSET
DAUD_CHAN3_VOICE_PRIORITY_OFFSET
DAUD_CHAN4_VOICE_PRIORITY_OFFSET
DAUD_CHAN5_VOICE_PRIORITY_OFFSET
DAUD_CHAN6_VOICE_PRIORITY_OFFSET
DAUD_CHAN7_VOICE_PRIORITY_OFFSET
DAUD_CHAN8_VOICE_PRIORITY_OFFSET
DAUD_CHAN9_VOICE_PRIORITY_OFFSET
DAUD_CHAN11_VOICE_PRIORITY_OFFSET
DAUD_CHAN12_VOICE_PRIORITY_OFFSET
DAUD_CHAN13_VOICE_PRIORITY_OFFSET
DAUD_CHAN14_VOICE_PRIORITY_OFFSET
DAUD_CHAN15_VOICE_PRIORITY_OFFSET
DAUD_CHAN16_VOICE_PRIORITY_OFFSET

The priority of a channel is represented by a range plus an offset. For example, DAUD_STANDARD_VOICE_PRIORITY combined with DAUD_CHAN10_VOICE_PRIORITY represents the highest priority within the

standard range. Priorities in the standard range are represented by defines in Dmusicc.h beginning with DAUD_CHAN1_DEF_VOICE_PRIORITY.

Channels that have the same priority value have equal priority, regardless of which channel group they belong to.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort8::SetChannelPriority

IDirectMusicPort8::GetFormat

Retrieves information about the wave format specified in the **DMUS_PORTPARAMS8** structure passed to **IDirectMusic8::CreatePort**, and the recommended size of the buffer to use for wave output. The information can be used to create a compatible DirectSound buffer for the port.

```
HRESULT GetFormat(
  LPWAVEFORMATEX pWaveFormatEx,
  LPDWORD pdwWaveFormatExSize
  LPDWORD pdwBufferSize
);
```

Parameters

pWaveFormatEx
Address of the **WAVEFORMATEX** structure that receives information about the format. This value can be NULL. See Remarks.

pdwWaveFormatExSize
Address of a variable that specifies or receives the size of the structure. See Remarks.

pdwBufferSize
Address of a variable that receives the recommended size of the DirectSound buffer.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK.

If it fails, the method can return E_POINTER.

Remarks

The **WAVEFORMATEX** structure can have a variable length that depends on the details of the format. Before retrieving the format description, the application should query the synthesizer object for the size of the format by calling this method and specifying NULL for the *pWaveFormatEx* parameter. The size of the structure is returned in the variable pointed to by *pdwWaveFormatExSize*. The application can then allocate sufficient memory and call **GetFormat** again to retrieve the format description.

If *pWaveFormatEx* is not NULL, DirectMusic writes, at most, *pdwWaveFormatExSize* bytes to the structure.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort8::SetDirectSound

IDirectMusicPort8::GetLatencyClock

Retrieves an **IReferenceClock** interface pointer to the port's latency clock. The latency clock specifies the nearest time in the future at which a message can be played on time. The latency clock is based on the DirectMusic master clock, which is set by using the **IDirectMusic8::SetMasterClock** method.

```
HRESULT GetLatencyClock(
  IReferenceClock** ppClock
);
```

Parameters

ppClock
Address of a variable that receives the latency clock's **IReferenceClock** interface pointer.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

In accordance with COM rules, **GetLatencyClock** increments the reference count of the returned interface. Therefore, the application must call **Release** on the **IReferenceClock** interface at some point.

Requirements

Header: Declared in dmusicc.h.

See Also

Latency and Bumper Time

IDirectMusicPort8::GetNumChannelGroups

Retrieves the number of channel groups on the port.

```
HRESULT GetNumChannelGroups(
  LPDWORD pdwChannelGroups
);
```

Parameters

pdwChannelGroups
Address of a variable that receives the number of channel groups.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG
E_NOTIMPL
E_OUTOFMEMORY

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort8::SetNumChannelGroups, Channels

IDirectMusicPort8::GetRunningStats

Retrieves information about the state of the synthesizer.

```
HRESULT GetRunningStats(
  LPDMUS_SYNTHSTATS pStats
);
```

Parameters

pStats

Address of a **DMUS_SYNTHSTATS8** structure that receives running statistics of the synthesizer. The **dwSize** member of this structure must be properly initialized before the method is called.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER

E_INVALIDARG

E_NOTIMPL

Remarks

Some hardware synthesizers might continue to report running statistics even though the port has been deactivated.

Requirements

Header: Declared in dmusicc.h.

IDirectMusicPort8::PlayBuffer

Cues a buffer for playback by the port.

```
HRESULT PlayBuffer(
  LPDIRECTMUSICBUFFER pBuffer
);
```

Parameters

pBuffer

Address of an **IDirectMusicBuffer8** interface pointer of the buffer to be added to the port's playback queue.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG
E_NOTIMPL
E_OUTOFMEMORY

Remarks

The buffer is in use by the system only for the duration of this method and can be reused after the method returns.

If no start time has been set by using the **IDirectMusicBuffer8::SetStartTime** method, the start time is the time of the earliest event in the buffer, as set by the **IDirectMusicBuffer8::PackStructured** or the **IDirectMusicBuffer8::PackUnstructured** method.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer8, **IDirectMusic8::CreateMusicBuffer**

IDirectMusicPort8::Read

Fills a buffer with incoming MIDI data. The method should be called with new buffer objects until no more data is available to be read.

```
HRESULT Read(
  LPDIRECTMUSICBUFFER pBuffer
);
```

Parameters

pBuffer
Address of the **IDirectMusicBuffer8** interface pointer of the buffer object to be filled with the incoming MIDI data.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE.

If it fails, the method can return one of the following error values:

E_POINTER
E_NOTIMPL

Remarks

When there is no more data to read, the method returns S_FALSE.

Requirements

Header: Declared in dmusicc.h.

See Also

Capturing MIDI

IDirectMusicPort8::SetChannelPriority

Sets the priority of a MIDI channel. For an overview, see Channels.

```
HRESULT SetChannelPriority(
  DWORD  dwChannelGroup,
  DWORD  dwChannel,
  DWORD  dwPriority
);
```

Parameters

dwChannelGroup
Group that the channel is in. This value must be 1 or greater.

dwChannel
Index of the channel on the group.

dwPriority
The priority ranking. See Remarks for **IDirectMusicPort8::GetChannelPriority**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL

E_INVALIDARG
E_OUTOFMEMORY
E_NOTIMPL

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort8::GetChannelPriority,
DMUS_CHANNEL_PRIORITY_PMSG.

IDirectMusicPort8::SetDirectSound

Overrides the default DirectSound object or buffer, or both, to which a port's wave data is streamed. This method is also used to disconnect the port from DirectSound.

```
HRESULT SetDirectSound(
  LPDIRECTSOUND pDirectSound,
  LPDIRECTSOUNDBUFFER pDirectSoundBuffer
);
```

Parameters

pDirectSound
Address of the **IDirectSound8** interface of the DirectSound object to which the port is to be connected, or NULL to disconnect and release the existing DirectSound object.

pDirectSoundBuffer
Address of the **IDirectSoundBuffer8** interface to connect the port to. This value can be NULL, and must be NULL if *pDirectSound* is NULL. This parameter is not used if the port is the Microsoft software synthesizer in DirectX 8.0 or later.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_NOBUFFERCONTROL. See Remarks.

If it fails, the method can return one of the following error values:

DMUS_E_ALREADY_ACTIVATED
E_INVALIDARG

Remarks

If a valid pointer is passed in *pDirectSoundBuffer*, the method returns DMUS_S_NOBUFFERCONTROL if control changes in the buffer such as pan and volume do not affect DirectMusic playback. This affects only Windows Driver Model (WDM) ports.

When the port is activated, the primary DirectSound buffer is upgraded, if necessary, to support the sample rate and channel information for this port (specified in the **DMUS_PORTPARAMS8** structure passed to **IDirectMusic8::CreatePort**).

The buffer pointed to by *pDirectSoundBuffer* must be a secondary streaming buffer with a format that matches the sample rate and channel information for this port. If this parameter is NULL, an appropriate **IDirectSoundBuffer** instance is created internally.

Neither the **IDirectSound** nor the **IDirectSoundBuffer** can be changed after the port has been activated.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort8::Activate, **IDirectMusicPort8::GetFormat**

IDirectMusicPort8::SetNumChannelGroups

Changes the number of channel groups that the application needs on the port.

```
HRESULT SetNumChannelGroups(
  DWORD dwChannelGroups
);
```

Parameters

dwChannelGroups
Number of channel groups on this port that the application wants to allocate.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG

E_NOTIMPL
E_OUTOFMEMORY

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort8::GetNumChannelGroups, Channels

IDirectMusicPort8::SetReadNotificationHandle

Specifies an event that is to be set when MIDI messages are available to be read with the **IDirectMusicPort8::Read** method. The event is signaled whenever new data is available. To turn off event notification, call **SetReadNotificationHandle** with a NULL value for the *hEvent* parameter.

```
HRESULT SetReadNotificationHandle(
  HANDLE hEvent
);
```

Parameters

hEvent
Event handle obtained from a call to the Win32 **CreateEvent** function. It is the application's responsibility to close this handle after the port has been released.

Return Values

If it succeeds, the method returns S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_DMUSIC_RELEASED
E_NOTIMPL

Remarks

A return value of E_NOTIMPL can mean the port is not an input port.

Requirements

Header: Declared in dmusicc.h.

See Also

Capturing MIDI

IDirectMusicPort8::UnloadInstrument

Unloads a previously downloaded DLS instrument.

```
HRESULT UnloadInstrument(
  IDirectMusicDownloadedInstrument *pDownloadedInstrument
);
```

Parameters

pDownloadedInstrument
Address of an **IDirectMusicDownloadedInstrument8** interface, obtained when the instrument was downloaded by calling the **IDirectMusicPort8::DownloadInstrument** method.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_DOWNLOADED_TO_PORT
E_POINTER
E_NOTIMPL

Remarks

This method must be called to free memory allocated by **IDirectMusicPort8::DownloadInstrument**.

Requirements

Header: Declared in dmusicc.h.

See Also

Working with Instruments

IDirectMusicPortDownload8

The **IDirectMusicPortDownload8** interface allows an application to communicate directly with a port that supports DLS downloading and to download memory chunks

directly to the port. The interface is used primarily by authoring applications that edit DLS instruments directly. For an overview, see Low-Level DLS.

To obtain the **IDirectMusicPortDownload8** interface, call the **IDirectMusicPort8::QueryInterface** method, passing in IID_IDirectMusicPortDownload8 as the interface GUID. If the port does not support DLS downloading, this call might fail.

The methods of the **IDirectMusicPortDownload8** interface can be grouped as follows:

Buffer management	**AllocateBuffer**
	GetAppend
	GetBuffer
	GetDLId
Loading	**Download**
	Unload

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The LPDIRECTMUSICPORTDOWNLOAD8 type is defined as a pointer to this interface.

```
typedef IDirectMusicPortDownload8 *LPDIRECTMUSICPORTDOWNLOAD8;
```

Requirements

Header: Declared in dmusicc.h.

IDirectMusicPortDownload8::AllocateBuffer

Allocates a chunk of memory for downloading DLS data to the port and returns an **IDirectMusicDownload8** interface pointer that allows access to this buffer.

```
HRESULT AllocateBuffer(
  DWORD dwSize,
  IDirectMusicDownload** ppIDMDownload
);
```

Parameters

dwSize

Requested size of buffer.

ppIDMDownload
Address of a variable that receives the **IDirectMusicDownload8** interface pointer.

Return Values

If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_INVALIDARG
E_OUTOFMEMORY

Remarks

After a buffer has been allocated, its size cannot change.

The buffer is freed when the **IDirectMusicDownload8** interface is released.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPortDownload8::GetBuffer, Low-Level DLS

IDirectMusicPortDownload8::Download

Downloads a wave or instrument definition to the port. The memory must first be allocated by using the **IDirectMusicPortDownload8::AllocateBuffer** method.

```
HRESULT Download(
  IDirectMusicDownload* pIDMDownload
);
```

Parameters

pIDMDownload
Address of the **IDirectMusicDownload8** interface for the buffer.

Return Values

Return values are determined by the implementation of the port.

If the method succeeds, it returns S_OK.

If the method fails, it can return one of the following values:

E_POINTER
E_FAIL
DMUS_E_ALREADY_DOWNLOADED
DMUS_E_BADARTICULATION
DMUS_E_BADINSTRUMENT
DMUS_E_BADOFFSETTABLE
DMUS_E_BADWAVE
DMUS_E_BADWAVELINK
DMUS_E_BUFFERNOTSET
DMUS_E_NOARTICULATION
DMUS_E_NOTMONO
DMUS_E_NOTPCM
DMUS_E_UNKNOWNDOWNLOAD

Remarks

For more information on how to prepare the data to be downloaded, see Low-Level DLS.

After the memory has been downloaded, you cannot do anything more with it. To update the download, you must create a new buffer and assign it a new download ID obtained by using the **IDirectMusicPortDownload8::GetDLId** method, and then send it down.

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPortDownload8::**Unload**, **DMUS_DOWNLOADINFO**, **DMUS_OFFSETTABLE**

IDirectMusicPortDownload8::GetAppend

Retrieves the amount of memory that the port needs to be appended to the end of a download buffer. This extra memory can be used by the port to interpolate across a loop boundary.

```
HRESULT GetAppend(
  DWORD* pdwAppend
);
```

Parameters

pdwAppend
Address of a variable that receives the number of appended samples for which memory is required. The amount of memory can be calculated from the wave format.

Return Values

Return values are determined by the port implementation.

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_NOTIMPL

Requirements

Header: Declared in dmusicc.h.

IDirectMusicPortDownload8::GetBuffer

Retrieves the **IDirectMusicDownload8** interface pointer of a buffer whose unique identifier is known.

```
HRESULT GetBuffer(
  DWORD dwDLId,
  IDirectMusicDownload** ppIDMDownload
);
```

Parameters

dwDLId
Download identifier of the buffer. See **DMUS_DOWNLOADINFO**.

ppIDMDownload
Address of a variable that receives the **IDirectMusicDownload8** interface pointer for the buffer.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_INVALID_DOWNLOADID
DMUS_E_NOT_DOWNLOADED_TO_PORT

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPortDownload8::GetDLId, **IDirectMusicDownload8::GetBuffer**, Low-Level DLS.

IDirectMusicPortDownload8::GetDLId

Obtains sequential identifiers for one or more download buffers.

Every memory chunk downloaded to the synthesizer must have a unique identifier placed in its **DMUS_DOWNLOADINFO** structure. The **GetDLId** method guarantees that no two downloads have the same identifier.

```
HRESULT GetDLId(
  DWORD* pdwStartDLId,
  DWORD dwCount
);
```

Parameters

pdwStartDLId
Address of a variable that receives the first identifier.

dwCount
Number of identifiers to reserve. You might plan to download a whole series of chunks at once. Instead of calling **GetDLId** for each chunk, set *dwCount* to the number of chunks. **GetDLId** returns the first ID of the set, and the additional identifiers are automatically reserved up through **pdwStartDLId* plus *dwCount*. A subsequent call to **GetDLId** would skip past the reserved values.

Return Values

If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_INVALIDARG

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusicPortDownload8::GetBuffer, Low-Level DLS

IDirectMusicPortDownload8::Unload

Unloads a buffer that was previously downloaded by using **IDirectMusicPortDownload8::Download**.

```
HRESULT Unload(
  IDirectMusicDownload* pIDMDownload
);
```

Parameters

pIDMDownload
Address of the **IDirectMusicDownload8** interface for the buffer.

Return Values

Return values are determined by the port implementation.

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following values:

E_NOINTERFACE
DMUS_E_SYNTHNOTCONFIGURED

Requirements

Header: Declared in dmusicc.h.

IDirectMusicScript8

The **IDirectMusicScript8** interface represents a script that contains variables that can be set and retrieved by the application, and routines that can be called by the application.
Typically the interface is obtained by using **IDirectMusicLoader8::GetObject** to load a script file. The application then calls **IDirectMusicScript8::Init** to associate the script with the DirectMusicPerformance object that performs the actual playback.
The methods of the **IDirectMusicScript8** interface can be grouped as follows:

Initialization	**Init**

Calling routines	**CallRoutine**
Enumeration	**EnumRoutine**
	EnumVariable
Getting variables	**GetVariableObject**
	GetVariableNumber
	GetVariableVariant
Setting variables	**SetVariableNumber**
	SetVariableObject
	SetVariableVariant

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

See Also

Using Audio Scripts

IDirectMusicScript8::CallRoutine

Executes a routine in the script.

```
HRESULT CallRoutine(
  WCHAR *pwszRoutineName,
  DMUS_SCRIPT_ERRORINFO *pErrInfo
);
```

Parameters

pwszRoutineName
Name of the routine.

pErrInfo
Address of a **DMUS_SCRIPT_ERRORINFO** structure that receives information if an error occurs. Set this member to NULL if you do not want error information.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_GARBAGE_COLLECTED. See Garbage Collection.

If the method fails, return values can include the following:

DMUS_E_NOT_INIT

DMUS_E_SCRIPT_ERROR_IN_SCRIPT

DMUS_E_SCRIPT_ROUTINE_NOT_FOUND

E_POINTER

Remarks

Control does not return to the application until the routine finishes running.

Requirements

Header: Declared in dmusici.h.

IDirectMusicScript8::EnumRoutine

Retrieves the name of a routine in a script. This method might be used by music-authoring applications that need to enumerate all routines in a script.

```
HRESULT EnumRoutine(
  DWORD dwIndex,
  WCHAR *pwszName
);
```

Parameters

dwIndex
Zero-based index of the routine.

pwszName
Pointer to a string buffer that receives the name of the routine. Must contain at least MAX_PATH elements.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The routine was enumerated.
S_FALSE	There is no routine with the supplied index value.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

DMUS_S_STRING_TRUNCATED The name is longer than MAX_PATH.

If the method fails, return values can include the following:

DMUS_E_NOT_INIT
E_POINTER

Remarks

The following example, where *pScript* is a valid **IDirectMusicScript8** interface pointer, displays the indexes and names of all routines in the script:

```
WCHAR wszRoutineName[MAX_PATH];
int i = 0;
while (S_OK == pScript->EnumRoutine(i++, wszRoutineName))
   printf("Routine number %d is called %S\n", i, wszRoutineName);
```

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicScript8::EnumVariable

IDirectMusicScript8::EnumVariable

Retrieves the name of a variable in a script. This method might be used by music-authoring applications that need to enumerate all variables in a script.

```
HRESULT EnumVariable(
  DWORD dwIndex,
  WCHAR *pwszName
);
```

Parameters

dwIndex
Zero-based index of the variable.

pwszName
Address of a string buffer that receives the name of the variable. Must contain at least MAX_PATH elements.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The routine was enumerated.
S_FALSE	There is no routine with the supplied index value.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.
DMUS_S_STRING_TRUNCATED	The name is longer than MAX_PATH.

If the method fails, return values can include the following:

DMUS_E_NOT_INIT

Remarks

The following example, where *pScript* is a valid **IDirectMusicScript8** interface pointer, displays the indexes and names of all variables in the script:

```
WCHAR wszVariableName[MAX_PATH];
int i = 0;
while (S_OK == pScript->EnumVariable(i++, wszVariableName))
   printf("Variable number %d is called %S\n", i, wszVariableName);
```

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicScript8::EnumRoutine

IDirectMusicScript8::GetVariableNumber

Retrieves a 32-bit signed value from a variable declared in the script.

```
HRESULT GetVariableNumber(
  WCHAR *pwszVariableName,
  LONG *plValue,
  DMUS_SCRIPT_ERRORINFO *pErrInfo
);
```

Parameters

pwszVariableName
Name of the script variable.

plValue
Address of a variable that receives the value.

pErrInfo

Address of a **DMUS_SCRIPT_ERRORINFO** structure that receives information if an error occurs. Set this member to NULL if you do not want error information.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The value was retrieved.
S_FALSE	The variable does not exist in the script.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If the method fails, return values can include the following:

DISP_E_TYPEMISMATCH (See Winerror.h.)
DMUS_E_NOT_INIT
DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicScript8::SetVariableNumber

IDirectMusicScript8::GetVariableObject

Retrieves an object pointer from a variable declared in the script.

```
HRESULT GetVariableObject(
  WCHAR *pwszVariableName,
  REFIID riid,
  LPVOID FAR *ppv,
  DMUS_SCRIPT_ERRORINFO *pErrInfo
);
```

Parameters

pwszVariableName
Name of the script variable.

riid
Unique identifier of the interface. See the IID defines in Dmusici.h. All the standard interfaces have a defined identifier consisting of "IID_" plus the name

of the interface. For example, the identifier of **IDirectMusicTrack8** is IID_IDirectMusicTrack8.

ppv
Address of a variable that receives a pointer to the desired interface of the object.

pErrInfo
Address of a **DMUS_SCRIPT_ERRORINFO** structure that receives information if an error occurs. Set this member to NULL if you do not want error information.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The value was retrieved.
S_FALSE	The variable does not exist in the script.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If the method fails, return values can include the following:

DMUS_E_NOT_INIT
DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
E_NOINTERFACE
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicScript8::SetVariableObject

IDirectMusicScript8::GetVariableVariant

Retrieves a variant value from a variable declared in the script.

```
HRESULT GetVariableVariant(
  WCHAR *pwszVariableName,
  VARIANT *pvarValue,
  DMUS_SCRIPT_ERRORINFO *pErrInfo
);
```

Parameters

pwszVariableName
Name of the script variable.

pvarValue
Address of a variable that receives the value.

pErrInfo
Address of a **DMUS_SCRIPT_ERRORINFO** structure that receives information if an error occurs. Set this member to NULL if you do not want error information.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The value was retrieved.
S_FALSE	The variable does not exist in the script.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If the method fails, return values can include the following:

DMUS_E_NOT_INIT
DMUS_E_SCRIPT_CONTENT_READONLY
DMUS_E_SCRIPT_UNSUPPORTED_VARTYPE
DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicScript8::SetVariableVariant

IDirectMusicScript8::Init

Associates the script with the performance that will play the sounds.

```
HRESULT Init(
  IDirectMusicPerformance *pPerformance,
  DMUS_SCRIPT_ERRORINFO *pErrInfo
);
```

Parameters

pPerformance
Address of the **IDirectMusicPerformance8** interface of the performance object.

pErrInfo
Address of a **DMUS_SCRIPT_ERRORINFO** structure that receives information if an error occurs. Set this member to NULL if you do not want error information.

Requirements

Header: Declared in dmusici.h.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The script was initialized.
S_FALSE	The script has already been attached to a different performance.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If the method fails, return values can include the following:

DMUS_E_NOT_INIT
DMUS_E_SCRIPT_CONTENT_READONLY
DMUS_E_SCRIPT_ERROR_IN_SCRIPT
DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
E_POINTER
E_NOINTERFACE

IDirectMusicScript8::SetVariableNumber

Assigns a 32-bit signed value to a variable declared in the script.

```
HRESULT SetVariableNumber(
  WCHAR *pwszVariableName,
  LONG lValue,
  DMUS_SCRIPT_ERRORINFO *pErrInfo
);
```

Parameters

pwszVariableName
Name of the script variable.

lValue
Value to assign to the variable.

pErrInfo

Address of a **DMUS_SCRIPT_ERRORINFO** structure that receives information if an error occurs. Set this member to NULL if you do not want error information.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The value was set.
S_FALSE	The variable does not exist in the script.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If the method fails, return values can include the following:

DMUS_E_NOT_INIT
DMUS_E_SCRIPT_CONTENT_READONLY
DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicScript8::GetVariableNumber

IDirectMusicScript8::SetVariableObject

Assigns an object interface pointer to a variable declared in the script.

```
HRESULT SetVariableObject(
  WCHAR *pwszVariableName,
  IUnknown *punkValue,
  DMUS_SCRIPT_ERRORINFO *pErrInfo
);
```

Parameters

pwszVariableName
Name of the script variable.

punkValue
Interface pointer to assign to the variable. This can be an interface of one of the DirectMusic objects supported by the script engine, such as

IDirectMusicSegment8, or an interface of any other object that implements the **IDispatch** interface.

pErrInfo
Address of a **DMUS_SCRIPT_ERRORINFO** structure that receives information if an error occurs. Set this member to NULL if you do not want error information.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The value was set.
S_FALSE	The variable does not exist in the script.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If the method fails, return values can include the following:

DMUS_E_NOT_INIT
DMUS_E_SCRIPT_CONTENT_READONLY
DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicScript8::GetVariableObject

IDirectMusicScript8::SetVariableVariant

Assigns a variant value to a variable declared in the script.

```
HRESULT SetVariableVariant(
  WCHAR *pwszVariableName,
  VARIANT varValue,
  BOOL fSetRef,
  DMUS_SCRIPT_ERRORINFO *pErrInfo
);
```

Parameters

pwszVariableName
Name of the script variable.

varValue
Value to assign to the variable.

fSetRef
TRUE if the variable is to be set by reference, FALSE if by value. Only objects can be set by reference. This flag should always be TRUE for DirectMusic objects and FALSE for other variants.

pErrInfo
Address of a **DMUS_SCRIPT_ERRORINFO** structure that receives information if an error occurs. Set this member to NULL if you do not want error information.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The value was set.
S_FALSE	The variable does not exist in the script.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If the method fails, return values can include the following:

DMUS_E_NOT_INIT
DMUS_E_SCRIPT_CONTENT_READONLY
DMUS_E_SCRIPT_NOT_A_REFERENCE
DMUS_E_SCRIPT_UNSUPPORTED_VARTYPE
DMUS_E_SCRIPT_VALUE_NOT_SUPPORTED
DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicScript8::GetVariableObject

IDirectMusicSegment8

The **IDirectMusicSegment8** interface represents a segment, a piece of music made up of multiple tracks. Most applications deal with musical data at the segment level. For an overview, see Using Segments.
The DirectMusicSegment object also supports the **IDirectMusicObject8** and **IPersistStream** interfaces for loading its data.

IDirectMusicSegment8 supersedes the **IDirectMusicSegment** interface and introduces new methods.

The methods of the **IDirectMusicSegment8** interface can be grouped as follows:

Instruments	**Download**
	Unload
Notification	**AddNotificationType**
	RemoveNotificationType
Parameters	**GetParam**
	SetParam
Timing and looping	**GetDefaultResolution**
	GetLength
	GetLoopPoints
	GetRepeats
	GetStartPoint
	SetDefaultResolution
	SetLength
	SetLoopPoints
	SetRepeats
	SetStartPoint
Tools	**GetGraph**
	SetGraph
Tracks	**GetTrack**
	GetTrackGroup
	InsertTrack
	RemoveTrack
	SetTrackConfig
Miscellaneous	**Clone**
	Compose
	GetAudioPathConfig
	InitPlay
	SetPChannelsUsed

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicSegment8::AddNotificationType

Adds a notification type. This method is called by the **IDirectMusicPerformance8::AddNotificationType** method, allowing the segment to respond to notifications. The segment calls each track's **IDirectMusicTrack8::AddNotificationType** method.

```
HRESULT AddNotificationType(
  REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType

Reference to (C++) or address of (C) the identifier of the notification type to add. For the defined types, see **DMUS_NOTIFICATION_PMSG**. Applications can also define their own types for custom tracks.

Return Values

If the method succeeds,Remarks

DirectSound does not initialize the contents of the buffer, and the application cannot assume that it contains silence.

If an attempt is made to create a buffer with the DSBCAPS_LOCHARDWARE flag on a system where hardware acceleration is not available, the method fails with DSERR_CONTROLUNAVAIL on Windows 95 and Windows 98. On Windows 2000, it returns DSERR_INVALIDCALL.

the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER

E_OUTOFMEMORY

Remarks

Segments cannot generate notifications of type GUID_NOTIFICATION_PERFORMANCE. To get notifications of this type, you must call **IDirectMusicPerformance8::AddNotificationType**.

Requirements

Header: Declared in dmusici.h.

See Also

Notification and Event Handling

IDirectMusicSegment8::Clone

Copies all or part of the segment and the tracks that it contains.

```
HRESULT Clone(
  MUSIC_TIME mtStart,
  MUSIC_TIME mtEnd,
  IDirectMusicSegment** ppSegment
);
```

Parameters

mtStart
Start of the part to copy. If less than 0 or greater than the length of the segment, 0 is used.

mtEnd
End of the part to copy. If this value is past the end of the segment, the segment is copied to the end. A value of 0 or anything less than *mtStart* also copies to the end.

ppSegment
Address of a variable that receives a pointer to the **IDirectMusicSegment** interface of the created segment. Use **QueryInterface** to obtain **IDirectMusicSegment8**. It is the caller's responsibility to call **Release** when finished with the segment.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if some tracks failed to copy.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY
E_POINTER

Remarks

Properties of the original segment, including start and loop points, number of repeats, and any toolgraph and default audiopath, are copied to the clone.

For style-based segments, if *mtStart* is greater than 0, it should be on a measure boundary.

Requirements

Header: Declared in dmusici.h.

IDirectMusicSegment8::Compose

Composes all tracks flagged as DMUS_TRACKCONFIG_COMPOSING and places the composed tracks in this segment or in a clone of this segment.

```
HRESULT Compose(
  MUSIC_TIME mtTime,
  IDirectMusicSegment* pFromSegment,
  IDirectMusicSegment* pToSegment,
  IDirectMusicSegment** ppComposedSegment
);
```

Parameters

mtTime
Value of type MUSIC_TIME that specifies the current time in *pFromSegment* at which to compose a transition. Set to 0 if *pFromSegment* is NULL.

pFromSegment
Pointer to an **IDirectMusicSegment8** interface that specifies the segment leading to a transition. This value is NULL if the calling segment is not a transition.

pToSegment
Pointer to the **IDirectMusicSegment8** interface that specifies the segment following a transition. This value is NULL if the calling segment is not a transition or if the transition is an ending.

ppComposedSegment
Address of a variable that receives the **IDirectMusicSegment8** interface pointer of the composed segment, or NULL if the calling segment is to be recomposed.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

DMUS_E_NOT_FOUND
E_FAIL
E_OUTOFMEMORY
E_POINTER

Remarks

If the pointer parameters are all NULL, the segment calls **IDirectMusicTrack8::Compose** on all its tracks. Any composing tracks search for other tracks necessary for composition; if a needed track is not found, DMUS_E_NOT_FOUND is returned.

If *ppComposedSegment* is not NULL, the method creates a clone of the original segment that contains the recomposed tracks. If either *pFromSegment* or *pToSegment* is not NULL, the calling segment is assumed to be a transition and might include tracks that contain only headers referring to one of the bracketing segments.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicTrack8::Compose, **IDirectMusicSegment8::SetTrackConfig**

IDirectMusicSegment8::Download

Downloads band data to a performance or audiopath.

```
HRESULT Download(
  IUnknown *pAudioPath
);
```

Parameters

pAudioPath

Pointer to the **IUnknown** interface of the performance or audiopath that receives the data.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method may return one of the following error values.

DMUS_E_NOT_FOUND

DMUS_E_TRACK_NOT_FOUND

E_POINTER

Remarks

All bands and wave data in the segment are downloaded.

Always call **IDirectMusicSegment8::Unload** before releasing the segment.

Requirements

Header: Declared in dmusici.h.

See Also

Using Bands

IDirectMusicSegment8::GetAudioPathConfig

Retrieves an object that represents an audiopath configuration embedded in the segment. The object can be passed to **IDirectMusicPerformance8::CreateAudioPath**.

```
HRESULT GetAudioPathConfig(
  IUnknown ** ppAudioPathConfig
);
```

Parameters

ppAudioPathConfig
Address of a variable that receives a pointer to the **IUnknown** interface of the audiopath configuration object.

Return Values

If the method succeeds, the return value is S_OK.

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

DMUS_E_NO_AUDIOPATH_CONFIG
E_POINTER

Requirements

Header: Declared in dmusici.h.

IDirectMusicSegment8::GetDefaultResolution

Retrieves the default resolution for synchronization.

```
HRESULT GetDefaultResolution(
  DWORD* pdwResolution
```

```
);
```

Parameters

pdwResolution
Address of a variable that receives the default resolution. See **DMUS_SEGF_FLAGS**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetDefaultResolution, Segment Timing

IDirectMusicSegment8::GetGraph

Retrieves the segment's toolgraph.

```
HRESULT GetGraph(
  IDirectMusicGraph** ppGraph
);
```

Parameters

ppGraph
Address of a variable that receives a pointer to the toolgraph.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
E_POINTER

Remarks

If there is no graph in the segment, the method returns DMUS_E_NOT_FOUND.

The reference count of the toolgraph is incremented.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetGraph

IDirectMusicSegment8::GetLength

Retrieves the length of the segment.

```
HRESULT GetLength(
  MUSIC_TIME* pmtLength
);
```

Parameters

pmtLength
Address of a variable that receives the segment's length in music time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

The method always returns 1 in **pmtLength* for segments created from wave files.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetLength

IDirectMusicSegment8::GetLoopPoints

Retrieves the start and end loop points inside the segment that repeat the number of times set by the **IDirectMusicSegment8::SetRepeats** method.

```
HRESULT GetLoopPoints(
  MUSIC_TIME* pmtStart,
```

MUSIC_TIME* *pmtEnd*
);

Parameters

pmtStart
Address of a variable that receives the start point of the loop.

pmtEnd
Address of a variable that receives the end point of the loop. A value of 0 indicates that the entire segment loops.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetLoopPoints, Segment Timing

IDirectMusicSegment8::GetParam

Retrieves data from a track inside this segment.

HRESULT GetParam(
REFGUID *rguidType*,
DWORD *dwGroupBits*,
DWORD *dwIndex*,
MUSIC_TIME *mtTime*,
MUSIC_TIME* *pmtNext*,
void* *pParam*
);

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data to obtain. See Standard Track Parameters.

dwGroupBits
Group that the desired track is in. Use 0xFFFFFFFF for all groups. For more information, see Identifying the Track.

dwIndex
Index of the track in the group identified by *dwGroupBits* from which to obtain the data, or DMUS_SEG_ANYTRACK to find the first track that contains the parameter.

mtTime
Time from which to obtain the data.

pmtNext
Address of a variable that receives the segment time (relative to *mtTime*) until which the data is valid. If this returns a value of 0, it means either that the data is always valid, or that it is unknown when it might become invalid. If this information is not needed, *pmtNext* can be set to NULL. See Remarks.

pParam
Address of an allocated structure in which the data is to be returned. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_GET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND
E_POINTER

Remarks

The data can become invalid before the time returned in **pmtNext* if another control segment is cued. For more information, see Control Segments.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, Performance Parameters

IDirectMusicSegment8::GetRepeats

Retrieves the number of times the looping portion of the segment is set to repeat.

```
HRESULT GetRepeats(
  DWORD* pdwRepeats
);
```

Parameters

pdwRepeats
Address of a variable that receives the number of times that the looping portion of the segment is set to repeat.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetRepeats

IDirectMusicSegment8::GetStartPoint

Retrieves the point within the segment at which it started or will start playing.

```
HRESULT GetStartPoint(
  MUSIC_TIME* pmtStart
);
```

Parameters

pmtStart
Address of a variable that receives the time within the segment at which it starts playing.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetStartPoint

IDirectMusicSegment8::GetTrack

Searches the list of tracks for the one with the supplied type, group, and index, and retrieves a pointer to the DirectMusicTrack object.

```
HRESULT GetTrack(
  REFGUID rguidType,
  DWORD dwGroupBits,
  DWORD dwIndex,
  IDirectMusicTrack** ppTrack
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the class identifier of the track to find. A value of GUID_NULL retrieves any track. For track identifiers, see Standard Track Types.

dwGroupBits
Track groups in which to scan for the track. A value of 0 is invalid. Each bit in *dwGroupBits* corresponds to a track group. To scan all tracks, regardless of groups, set this parameter to 0xFFFFFFFF.

dwIndex
Zero-based index into the list of tracks of type *rguidType* and in group *dwGroupBits* to return. If multiple groups are selected in *dwGroupBits*, this index indicates the *n*th track of type *rguidType* encountered in the union of the groups selected. See Remarks.

ppTrack
Address of a variable that receives a pointer to the track. The variable is set to NULL if the track is not found.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND

E_FAIL

E_POINTER

Remarks

To enumerate all tracks, use GUID_NULL for the *rguidType* and 0xFFFFFFFF for *dwGroupBits*. Call **GetTrack** starting with 0 for *dwIndex*, incrementing *dwIndex* until the method no longer returns a success code.

Tracks in segments created by DirectMusic Producer are not necessarily in the same order as they were in that application. Do not rely on *dwIndex* alone to find a particular track.

For more information on track groups, see Identifying the Track.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::InsertTrack

IDirectMusicSegment8::GetTrackGroup

Retrieves the group bits set on a track inside the segment.

```
HRESULT GetTrackGroup(
  IDirectMusicTrack* pTrack,
  DWORD* pdwGroupBits
);
```

Parameters

pTrack
Track for which to find the group bits.

pdwGroupBits
Address of a variable that receives the groups. Each bit in **pdwGroupBits* corresponds to a track group.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
E_INVALIDARG

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::InsertTrack, Identifying the Track

IDirectMusicSegment8::InitPlay

Initializes the play state. This method was for internal use and is not implemented in versions later than DirectX 7.0.

```
HRESULT InitPlay(
  IDirectMusicSegmentState** ppSegState,
  IDirectMusicPerformance* pPerformance,
  DWORD dwFlags
);
```

Parameters

ppSegState
Address of a variable that receives a pointer to the **IDirectMusicSegmentState8** interface that is created in response to this method call and is used to hold state data. It is returned with a reference count of 1, so a call to its **Release** method fully releases it.

pPerformance
Address of the **IDirectMusicPerformance8** interface. This is needed by the segment and segment state to call methods on the performance object.

dwFlags
DMUS_SEGF_FLAGS that modify the track's behavior.

Return Values

In DirectX 8.0 and later, the method returns E_NOTIMPL.

In earlier versions, if the method succeeds, the return value is S_OK. If it fails, it can return one of the following error values:

E_POINTER
E_OUTOFMEMORY

Requirements

Header: Declared in dmusici.h.

IDirectMusicSegment8::InsertTrack

Inserts the supplied track into the segment's list of tracks.

```
HRESULT InsertTrack(
```

```
  IDirectMusicTrack* pTrack,
  DWORD dwGroupBits
);
```

Parameters

pTrack
Track to add to the segment.

dwGroupBits
Group or groups into which to insert the track. This value cannot be 0.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_FAIL
E_INVALIDARG
E_OUTOFMEMORY
E_POINTER

Remarks

Tracks are put in groups to link them correctly. For example, a segment might contain two style tracks and two mute tracks. Each style track would be put in a different group, along with its associated mute track. For more information on track groups, see Identifying the Track.

If the segment is currently playing, the new track is not included in playback because the segment state was not initialized with the new track.

This method initializes the track. However, if the track data is subsequently changed, the application must initialize it again by calling **IDirectMusicTrack8::Init**.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::RemoveTrack,
IDirectMusicSegment8::GetTrackGroup

IDirectMusicSegment8::RemoveNotificationType

Removes a notification type. This method is called by the **IDirectMusicPerformance8::RemoveNotificationType** method, allowing the segment to remove notifications. The segment calls each track's **IDirectMusicTrack8::RemoveNotificationType** method.

```
HRESULT RemoveNotificationType(
  REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType
Reference to (C++) or address of (C) the identifier of the notification type to remove. (For the defined types, see **DMUS_NOTIFICATION_PMSG**.) Setting this value to GUID_NULL causes all notifications to be removed.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the notification type was not previously set.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

Notification and Event Handling

IDirectMusicSegment8::RemoveTrack

Removes a track from the segment's track list.

```
HRESULT RemoveTrack(
  IDirectMusicTrack* pTrack
);
```

Parameters

pTrack
Track to remove from the segment's track list.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the specified track is not in the track list.

If the method fails, the return value can be E_POINTER.

Remarks

The track is released when removed.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::InsertTrack

IDirectMusicSegment8::SetDefaultResolution

Sets the default resolution for synchronization.

```
HRESULT SetDefaultResolution(
  DWORD dwResolution
);
```

Parameters

dwResolution

Desired default resolution. This value can be 0 or one of the following members of the **DMUS_SEGF_FLAGS** enumeration:

DMUS_SEGF_MEASURE
DMUS_SEGF_BEAT
DMUS_SEGF_GRID

Return Values

The method returns S_OK.

Remarks

This method is used primarily by secondary segments (motifs) to define whether they are synchronized to the measure, beat, or grid resolutions.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::GetDefaultResolution, Segment Timing

IDirectMusicSegment8::SetGraph

Assigns a toolgraph to the segment.

```
HRESULT SetGraph(
  IDirectMusicGraph* pGraph
);
```

Parameters

pGraph
Toolgraph pointer. Can be set to NULL to clear the segment graph.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

Any messages flowing through tools in the current toolgraph are deleted.

The graph's reference count is incremented, so it is safe to release the original reference.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SetGraph, DirectMusic Tools

IDirectMusicSegment8::SetLength

Sets the length, in music time, of the segment. This method is usually called by the loader, which retrieves the segment length from the file and passes it to the segment object.

```
HRESULT SetLength(
  MUSIC_TIME mtLength
);
```

Parameters

mtLength
Desired length. Must be greater than 0.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
DMUS_E_OUT_OF_RANGE

Remarks

Neglecting to set a primary segment length can cause problems when cueing other primary segments with the DMUS_SEGF_QUEUE flag.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::GetLength

IDirectMusicSegment8::SetLoopPoints

Sets the start and end points of the part of the segment that repeats. It repeats the number of times set by the **IDirectMusicSegment8::SetRepeats** method.

```
HRESULT SetLoopPoints(
  MUSIC_TIME mtStart,
  MUSIC_TIME mtEnd
);
```

Parameters

mtStart
Point at which to begin the loop.

mtEnd

Point at which to end the loop. A value of 0 loops the entire segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_OUT_OF_RANGE.

Remarks

When the segment is played, it plays from the segment start time until *mtEnd*, then loops to *mtStart*, plays the looped portion the number of times set by **IDirectMusicSegment8::SetRepeats**, and then plays to the end.

The default values are set to loop the entire segment from beginning to end.

The method fails if *mtStart* is greater than or equal to the length of the segment, or if *mtEnd* is greater than the length of the segment. If *mtEnd* is 0, *mtStart* must be 0, as well.

This method does not affect any currently playing segment states created from this segment.

The loop points of a cached segment persist even if the segment is released, and then reloaded. To ensure that a segment is not subsequently reloaded from the cache, call **IDirectMusicLoader8::ReleaseObject** on it before releasing it.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::GetLoopPoints, Segment Timing

IDirectMusicSegment8::SetParam

Sets data on a track inside this segment.

```
HRESULT SetParam(
  REFGUID rguidType,
  DWORD dwGroupBits,
  DWORD dwIndex,
  MUSIC_TIME mtTime,
  void* pParam
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the type of data to set. See Standard Track Parameters.

dwGroupBits
Group that the desired track is in. Use 0xFFFFFFFF for all groups. For more information, see Identifying the Track.

dwIndex
Index of the track in the group identified by *dwGroupBits* in which to set the data, or DMUS_SEG_ALLTRACKS to set the parameter on all tracks in the group that contain the parameter.

mtTime
Time at which to set the data.

pParam
Address of a structure containing the data, or NULL if no data is required. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SetParam, **IDirectMusicSegment8::GetParam**, **IDirectMusicTrack8::SetParamEx**, Performance Parameters

IDirectMusicSegment8::SetPChannelsUsed

Sets the performance channels that this segment uses. This method is usually called by a track in the **IDirectMusicTrack8::Init** method to inform the segment of which channels the track uses.

HRESULT SetPChannelsUsed(

```
  DWORD dwNumPChannels,
  DWORD* paPChannels
);
```

Parameters

dwNumPChannels
Number of PChannels to set. This must be equal to the number of members in the array pointed to by *paPChannels*.

paPChannels
Address of an array of PChannels.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_OUTOFMEMORY
E_POINTER

Remarks

This method allows the performance to know which ports are being used by the segment so that it can determine the actual latency, rather than providing for the worst case.

Requirements

Header: Declared in dmusici.h.

See Also

Latency and Bumper Time, Channels

IDirectMusicSegment8::SetRepeats

Sets the number of times the looping portion of the segment is to repeat. By default, the entire segment is looped.

```
HRESULT SetRepeats(
  DWORD dwRepeats
);
```

Parameters

dwRepeats
Number of times that the looping portion of the segment is to repeat, or DMUS_SEG_REPEAT_INFINITE to repeat until explicitly stopped. A value of 0 specifies a single play with no repeats.

Return Values

The method returns S_OK.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::GetRepeats, **IDirectMusicSegment8::SetLoopPoints**, Segment Timing

IDirectMusicSegment8::SetStartPoint

Sets the point within the segment at which it will start playing.

```
HRESULT SetStartPoint(
  MUSIC_TIME mtStart
);
```

Parameters

mtStart
Point within the segment at which it is to start playing. Must be greater than or equal to zero and less than the length of the segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_OUT_OF_RANGE.

Remarks

The start point is the first point in the segment that can possibly be heard. However, the actual first point heard may be later, if the start point of the segment is aligned to a past time. For more information, see Segment Timing.

By default, the start point is 0, meaning that the segment starts from the beginning.

If the segment does not already have a length, **IDirectMusicSegment8::SetLength** must be called before this method.

The method does not affect any currently playing segment states created from this segment.

The start point of a cached segment persists even if the segment is released, and then reloaded. To ensure that a segment is not subsequently reloaded from the cache, call **IDirectMusicLoader8::ReleaseObject** on it before releasing it.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::GetStartPoint, **IDirectMusicSegmentState8::GetStartPoint**, **IDirectMusicSegment8::SetLength**, **IDirectMusicSegment8::SetLoopPoints**, Segment Timing

IDirectMusicSegment8::SetTrackConfig

Sets the configuration settings of a track.

```
HRESULT SetTrackConfig(
  REFGUID rguidTrackClassID,
  DWORD dwGroupBits,
  DWORD dwIndex,
  DWORD dwFlagsOn,
  DWORD dwFlagsOff
);
```

Parameters

rguidTrackClassID
Reference to (C++) or address of (C) the identifier of the track class. For a list of values, see Standard Track Types.

dwGroupBits
Groups to which the track belongs.

dwIndex
Index of the track within the group, or DMUS_SEG_ALLTRACKS to set the configuration of all tracks in the group.

dwFlagsOn
Configuration flags to set. See Remarks.

dwFlagsOff
Configuration flags to clear. See Remarks.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_TRACK_NOT_FOUND
E_INVALIDARG

Remarks

If you change a flag on a segment, subsequent instances of segment states inherit the change. However, segment states that are already playing do not change their behavior.

The following flags are defined.

DMUS_TRACKCONFIG_COMPOSING
Use this track to compose other tracks.

DMUS_TRACKCONFIG_CONTROL_ENABLED
Enable **IDirectMusicTrack8::GetParamEx**.

DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
When played in a controlling segment, override notification of primary segment tracks.

DMUS_TRACKCONFIG_CONTROL_PLAY
When played in a controlling segment, override playback of primary segment tracks.

DMUS_TRACKCONFIG_DEFAULT
The combination of DMUS_TRACKCONFIG_CONTROL_ENABLED | DMUS_TRACKCONFIG_PLAY_ENABLED | DMUS_TRACKCONFIG_NOTIFICATION_ENABLED.

DMUS_TRACKCONFIG_FALLBACK
The track tries to get parameters from this segment if the primary and control segments don't return the requested information.

DMUS_TRACKCONFIG_LOOP_COMPOSE
Regenerate data each time the track repeats.

DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
Enable notifications.

DMUS_TRACKCONFIG_OVERRIDE_ALL
The track tries to get parameters from this segment before the control and primary segment.

DMUS_TRACKCONFIG_OVERRIDE_PRIMARY
The track tries to get parameters from this segment before the primary segment.

DMUS_TRACKCONFIG_PLAY_CLOCKTIME
Play in clock time, not music time.

DMUS_TRACKCONFIG_PLAY_COMPOSE

Regenerate data each time the track starts playing.

DMUS_TRACKCONFIG_PLAY_ENABLED
Enable track to send messages.

DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
In composing transitions, get track information from the current place in the first segment.

DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
In composing transitions, get track information from the start of the first segment.

DMUS_TRACKCONFIG_TRANS1_TOSEGSTART
In composing transitions, get track information from the start of the second segment.

The following table shows which track configuration flags are valid for standard tracks.

Band

DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED
DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
DMUS_TRACKCONFIG_TRANS1_TOSEGSTART

Chord

DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
DMUS_TRACKCONFIG_TRANS1_TOSEGSTART

Chordmap

DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
DMUS_TRACKCONFIG_TRANS1_TOSEGSTART

Command

DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
DMUS_TRACKCONFIG_NOTIFICATION_ENABLED

DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
DMUS_TRACKCONFIG_TRANS1_TOSEGSTART

Lyrics
DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED

Marker:
DMUS_TRACKCONFIG_CONTROL_ENABLED

Melody formulation. Not implemented in DirectX 8.0.
DMUS_TRACKCONFIG_COMPOSING
DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_LOOP_COMPOSE
DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
DMUS_TRACKCONFIG_PLAY_COMPOSE
DMUS_TRACKCONFIG_PLAY_ENABLED
DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
DMUS_TRACKCONFIG_TRANS1_TOSEGSTART

Motif
DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_FALLBACK
DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
DMUS_TRACKCONFIG_OVERRIDE_ALL
DMUS_TRACKCONFIG_OVERRIDE_PRIMARY
DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED

Mute
DMUS_TRACKCONFIG_CONTROL_ENABLED

Parameter control
DMUS_TRACKCONFIG_PLAY_CLOCKTIME

DMUS_TRACKCONFIG_PLAY_ENABLED

Pattern
DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_FALLBACK
DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
DMUS_TRACKCONFIG_OVERRIDE_ALL
DMUS_TRACKCONFIG_OVERRIDE_PRIMARY
DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED

Script
DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED

Segment Trigger
DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED

Sequence
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_FALLBACK
DMUS_TRACKCONFIG_OVERRIDE_ALL
DMUS_TRACKCONFIG_OVERRIDE_PRIMARY
DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED

Signpost
DMUS_TRACKCONFIG_COMPOSING
DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_LOOP_COMPOSE
DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
DMUS_TRACKCONFIG_PLAY_COMPOSE
DMUS_TRACKCONFIG_PLAY_ENABLED
DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
DMUS_TRACKCONFIG_TRANS1_TOSEGSTART

Style
DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_FALLBACK
DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
DMUS_TRACKCONFIG_OVERRIDE_ALL
DMUS_TRACKCONFIG_OVERRIDE_PRIMARY
DMUS_TRACKCONFIG_PLAY_ENABLED
DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
DMUS_TRACKCONFIG_TRANS1_TOSEGSTART

Sysex
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED

Tempo
DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED
DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
DMUS_TRACKCONFIG_TRANS1_TOSEGSTART

Time signature
DMUS_TRACKCONFIG_CONTROL_ENABLED
DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
DMUS_TRACKCONFIG_PLAY_ENABLED

Wave
DMUS_TRACKCONFIG_CONTROL_PLAY
DMUS_TRACKCONFIG_FALLBACK
DMUS_TRACKCONFIG_OVERRIDE_ALL
DMUS_TRACKCONFIG_OVERRIDE_PRIMARY

DMUS_TRACKCONFIG_PLAY_CLOCKTIME
DMUS_TRACKCONFIG_PLAY_ENABLED

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegmentState8::SetTrackConfig, Self-Controlling Segments, Track Configuration

IDirectMusicSegment8::Unload

Unloads instrument data from a performance or audiopath.

```
HRESULT Unload(
  IUnknown *pAudioPath
);
```

Parameters

pAudioPath
Pointer to the **IUnknown** interface of the performance or audiopath from which to unload the instrument data.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

DMUS_E_TRACK_NOT_FOUND
E_POINTER

Remarks

The method succeeds even if no data was previously downloaded.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::Download

IDirectMusicSegmentState8

When the **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx** method is called, the performance engine generates a segment-state object that tracks the state of the playing segment. It also provides the application with a handle to the segment, in the form of the **IDirectMusicSegmentState8** interface, which can be used to track the playback status of the segment. This method can also be used directly to stop playback or remove the segment from the performance, using methods of **IDirectMusicPerformance8**.

IDirectMusicSegmentState8 supersedes **IDirectMusicSegmentState** and adds new methods.

The interface has the following methods:

Information	**GetObjectInPath**
	GetRepeats
	GetSeek
	GetSegment
	GetStartPoint
	GetStartTime
Track Configuration	**SetTrackConfig**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicSegmentState8::GetObjectInPath

Retrieves an interface for an object in the audiopath on which this segment state is playing.

```
HRESULT GetObjectInPath(
  DWORD dwPChannel,
  DWORD dwStage,
  DWORD dwBuffer,
  REFGUID guidObject,
```

DWORD *dwIndex*,
REFGUID *iidInterface*,
void ** *ppObject*
);

Parameters

dwPChannel
Performance channel to search, or DMUS_PCHANNEL_ALL to search all channels. The first channel is numbered 0. See Remarks.

dwStage
Stage in the path. Can be one of the following values.

DMUS_PATH_AUDIOPATH
The audiopath on which the segment state is playing.

DMUS_PATH_AUDIOPATH_GRAPH
The audiopath toolgraph. One is created if none exists.

DMUS_PATH_AUDIOPATH_TOOL
A tool from the audiopath toolgraph.

DMUS_PATH_BUFFER
A DirectSound buffer.

DMUS_PATH_BUFFER_DMO
A DMO in the buffer. For information on DMO interfaces, see **IDirectSoundBuffer8::GetObjectInPath**.

DMUS_PATH_MIXIN_BUFFER
A global mix-in buffer.

DMUS_PATH_MIXIN_BUFFER_DMO
A DMO in a global mix-in buffer.

DMUS_PATH_PERFORMANCE
The performance.

DMUS_PATH_PERFORMANCE_GRAPH
The performance toolgraph. One is created if none exists.

DMUS_PATH_PERFORMANCE_TOOL
A tool in the performance graph.

DMUS_PATH_PORT
The synthesizer.

DMUS_PATH_PRIMARY_BUFFER
The primary buffer.

DMUS_PATH_SEGMENT
The segment that owns this segment state.

DMUS_PATH_SEGMENT_GRAPH
The segment toolgraph. One is created if none exists. See Remarks.

DMUS_PATH_SEGMENT_TOOL
A tool from the segment graph. See Remarks.

DMUS_PATH_SEGMENT_TRACK
A track from the segment. See Remarks.

dwBuffer
If *dwStage* is DMUS_PATH_BUFFER_DMO or DMUS_PATH_MIXIN_BUFFER_DMO, the index of the buffer in which that DMO resides. If *dwStage* is DMUS_PATH_BUFFER or DMUS_PATH_MIXIN_BUFFER, the index of the buffer. Otherwise must be 0.

guidObject
Class identifier of the objector GUID_All_Objects to search for an object of any class. This parameter is ignored if only a single class of object can exist at the stage specified by *dwStage*, and can be set to GUID_NULL.

dwIndex
Index of the object in the list of matching objects. Set to 0 to find the first matching object. If *dwStage* is DMUS_PATH_BUFFER or DMUS_PATH_MIXIN_BUFFER, this parameter is ignored, and the buffer index is specified by *dwBuffer*.

iidInterface
Identifier of the desired interface, such as IID_IDirectMusicGraph.

ppObject
Address of a variable that receives a pointer to the requested interface.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_GARBAGE_COLLECTED. See Garbage Collection.

If it fails, the method can return one of the following error values.

DMUS_E_NOT_FOUND

E_INVALIDARG

E_NOINTERFACE

E_OUTOFMEMORY

Remarks

The value in *dwPChannel* must be 0 for any stage that is not channel-specific. Objects in the following stages are channel-specific and can be retrieved by setting a channel number or DMUS_PCHANNEL_ALL in *dwPChannel*:

DMUS_PATH_AUDIOPATH_TOOL
DMUS_PATH_BUFFER
DMUS_PATH_BUFFER_DMO
DMUS_PATH_PERFORMANCE_TOOL
DMUS_PATH_PORT
DMUS_PATH_SEGMENT_TOOL

The precedence of the parameters in filtering out unwanted objects is as follows:

1. *dwStage*.
2. *guidObject*. If this value is not GUID_All_Objects, only objects whose class identifier equals *guidObject* are searched. However, this parameter is ignored when only a single class of object can exist at the specified stage.
3. *dwPChannel*. If the stage is channel-specific and this value is not DMUS_PCHANNEL_ALL, only objects on the channel are searched.
4. *dwBuffer*. This is used only if *dwStage* is DMUS_PATH_BUFFER, DMUS_PATH_MIXIN_BUFFER, DMUS_PATH_BUFFER_DMO, or DMUS_PATH_MIXIN_BUFFER_DMO.
5. *dwIndex*. Note that tracks in segments created by DirectMusic Producer are not necessarily in the same order as they were in that application. Do not rely on *dwIndex* alone to find a particular track at stage DMUS_PATH_SEGMENT_TRACK.

If a matching object is found but the interface specified by *iidInterface* cannot be obtained, the method fails.

The object returned when DMUS_PATH_SEGMENT_GRAPH or DMUS_PATH_SEGMENT_TOOL is specified in *dwStage* might not be the same one returned for a different segment state based on the same segment. When a segment is played, its toolgraph is copied and any tools that support the **IDirectMusicTool8::Clone** method are also cloned.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicAudioPath8::GetObjectInPath, Retrieving Objects from an Audiopath

IDirectMusicSegmentState8::GetRepeats

Returns the number of times that the looping portion of the segment is set to repeat.

```
HRESULT GetRepeats(
  DWORD* pdwRepeats
);
```

Parameters

pdwRepeats

Address of a variable that receives the repeat count. A value of 0 indicates that the segment is to play through only once, with no portion repeated.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetRepeats

IDirectMusicSegmentState8::GetSeek

Retrieves the seek pointer in the segment state. This is the value that is passed in the *mtStart* parameter of **IDirectMusicTrack8::Play** the next time that method is called. It does not take into account looping and repeating; if the entire segment state repeats to the beginning, the seek pointer is reset to 0.

```
HRESULT GetSeek(
  MUSIC_TIME* pmtSeek
);
```

Parameters

pmtSeek
Address of a variable that receives the seek pointer.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be E_POINTER.

Requirements

Header: Declared in dmusici.h.

IDirectMusicSegmentState8::GetSegment

Returns a pointer to the segment that owns this segment state.

```
HRESULT GetSegment(
```

IDirectMusicSegment** *ppSegment*
);

Parameters

ppSegment
Address of a variable that receives a pointer to the **IDirectMusicSegment** interface. Use **QueryInterface** to obtain **IDirectMusicSegment8** interface.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

DMUS_E_NOT_FOUND
E_POINTER

Remarks

The pointer returned in *ppSegment* must be released by the application.

Requirements

Header: Declared in dmusici.h.

IDirectMusicSegmentState8::GetStartPoint

Returns the offset into the segment at which play will begin or began.

HRESULT GetStartPoint(
MUSIC_TIME * *pmtStart*
);

Parameters

pmtStart
Address of a variable that receives the music-time offset from the start of the segment at which the segment state initially played or will play.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

The start point is the first point in the segment that can possibly be heard. However, the actual first point heard may be later, if the start point of the segment is aligned to a past time. For more information, see Segment Timing.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetStartPoint,
IDirectMusicSegmentState8::GetStartTime

IDirectMusicSegmentState8::GetStartTime

Gets the performance time at which the segment started or will start playing.

```
HRESULT GetStartTime(
  MUSIC_TIME* pmtStart
);
```

Parameters

pmtStart
Address of a variable that receives the music-time offset stored in this segment state.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

The time retrieved by this method is the resolved play time within the performance where the segment start time was cued. For more information, see Segment Timing.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetStartPoint, **IDirectMusicSegment8::GetStartPoint**, **IDirectMusicSegmentState8::GetStartPoint**

IDirectMusicSegmentState8::SetTrackConfig

Sets the configuration settings of a track.

```
HRESULT SetTrackConfig(
  REFGUID rguidTrackClassID,
  DWORD dwGroupBits,
  DWORD dwIndex,
  DWORD dwFlagsOn,
  DWORD dwFlagsOff
);
```

Parameters

rguidTrackClassID
Reference to (C++) or address of (C) the identifier of the track class. For a list of values, see Standard Track Types.

dwGroupBits
Groups to which the track belongs.

dwIndex
Index of the track within the group, or DMUS_SEG_ALLTRACKS to set the configuration of all tracks in the group.

dwFlagsOn
Configuration flags to set. For a list of values, see **IDirectMusicSegment8::SetTrackConfig**.

dwFlagsOff
Configuration flags to clear.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_TRACK_NOT_FOUND
E_INVALIDARG

Remarks

Any change in configuration takes effect after prepare time. If you want the change to take effect immediately, call **IDirectMusicPerformance8::Invalidate**.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::SetTrackConfig, Self-Controlling Segments, Track Configuration

IDirectMusicSong8

Not implemented in DirectX 8.0.

The **IDirectMusicSong8** interface represents a song, which is a collection of segments based on common source material.

The interface has the following methods.

Instruments	**Download**
	Unload
Segment retrieval	**EnumSegment**
	GetSegment
Miscellaneous	**Compose**
	GetAudioPathConfig
	GetParam

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicSong8::Compose

Not implemented in DirectX 8.0.

Creates playable segments from the source material in the song.

```
HRESULT Compose();
```

Parameters

None.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The song was composed.
S_FALSE	There were no tracks to be composed.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If it fails, the method can return one of the following error values.

DMUS_E_TRACK_NOT_FOUND
E_INVALIDARG
E_POINTER
E_OUTOFMEMORY

Remarks

The method finds all composing tracks in all segments in the song, joins tracks with the same class identifier and group bits into master composing tracks, composes new tracks from the master composing tracks in priority order, breaks these new tracks up so that they correspond to the original tracks, and inserts the smaller composed tracks into segments of the song in the appropriate places.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::Compose, **IDirectMusicTrack8::Compose**

IDirectMusicSong8::Download

Not implemented in DirectX 8.0.

Downloads band data to a performance or audiopath.

```
HRESULT Download(
  IUnknown *pAudioPath
);
```

Parameters

pAudioPath
Pointer to the **IUnknown** interface of the performance or audiopath that receives the instrument data.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_PARTIALDOWNLOAD. For more information, see the Remarks for **IDirectMusicBand8::Download**.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_NOINTERFACE
E_OUTOFMEMORY
E_POINTER

Remarks

All bands in all segments of the song are downloaded.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSong8::Unload

IDirectMusicSong8::EnumSegment

Not implemented in DirectX 8.0.

Retrieves one of the segments that make up the song, by index.

```
HRESULT EnumSegment(
  DWORD dwIndex,
  IDirectMusicSegment **ppSegment
);
```

Parameters

dwIndex
Index of the segment within the song.

ppSegment

Address of a variable that receives an **IDirectMusicSegment** interface pointer if a matching segment is found. Use **QueryInterface** to obtain **IDirectMusicSegment8**.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The segment was enumerated.
S_FALSE	There was no segment at *dwIndex*.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

The method returns S_OK, or S_FALSE if *dwIndex* does not match any of the segments in the song.

If it fails, the method may return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSong8::GetSegment

IDirectMusicSong8::GetAudioPathConfig

Not implemented in DirectX 8.0.

Retrieves an object that represents the audiopath configuration for the song.

```
HRESULT GetAudioPathConfig(
  IUnknown ** ppAudioPathConfig
);
```

Parameters

ppAudioPathConfig

Address of a variable that receives a pointer to the **IUnknown** interface of the audiopath configuration object.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_GARBAGE_COLLECTED. See Garbage Collection.

If it fails, the method can return one of the following error values:

DMUS_E_NO_AUDIOPATH_CONFIG

E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::GetAudioPathConfig

IDirectMusicSong8::GetParam

Not implemented in DirectX 8.0.

Retrieves data from a track inside this song.

```
HRESULT GetParam(
  REFGUID rguidType,
  DWORD dwGroupBits,
  DWORD dwIndex,
  MUSIC_TIME mtTime,
  MUSIC_TIME* pmtNext,
  void* pParam
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data to obtain. See Standard Track Parameters.

dwGroupBits
Group that the desired track is in. Set this value to 0xFFFFFFFF for all groups. See Remarks.

dwIndex
Index of the track in the group from which to obtain the data.

mtTime
Time from which to obtain the data, in performance time.

pmtNext
Address of a variable that receives the time, relative to *mtTime*, until which the data is valid. If this returns a value of zero, either the data is always valid, or it is not known when it might become invalid. If this information is not needed, *pmtNext* can be set to NULL.

pParam
Address of a buffer that receives the parameter data.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_GARBAGE_COLLECTED. See Garbage Collection.

If it fails, the method can return one of the following error values:

DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND
E_POINTER

Requirements

Header: Declared in dmusici.h.

IDirectMusicSong8::GetSegment

Not implemented in DirectX 8.0.

Retrieves one of the segments that make up the song, by name.

```
HRESULT GetSegment(
  WCHAR pwzName,
  IDirectMusicSegment **ppSegment
);
```

Parameters

pwzName
Name of the segment, or NULL to retrieve the first segment.

ppSegment
Address of a variable that receives an **IDirectMusicSegment** interface pointer if a matching segment is found. Use **QueryInterface** to obtain **IDirectMusicSegment8**.

Return Values

If the method succeeds, one of the following success codes is returned:

S_OK	The segment was retrieved.
S_FALSE	*pwzName* does not match any segment.
DMUS_S_GARBAGE_COLLECTED	See Garbage Collection.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSong8::EnumSegment

IDirectMusicSong8::Unload

Not implemented in DirectX 8.0.

Unloads instrument data from an audiopath.

```
HRESULT Unload(
  IUnknown *pAudioPath
);
```

Parameters

pAudioPath
Pointer to the **IUnknown** interface of the audiopath from which to unload the instrument data.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_GARBAGE_COLLECTED. See Garbage Collection.

If it fails, the method may return E_POINTER.

Requirements

Header: Declared in dmusici.h.

Remarks

The method succeeds even if no data was previously downloaded.

See Also

IDirectMusicSong8::Download

IDirectMusicStyle8

The **IDirectMusicStyle8** interface provides access to a style object. The style object provides the performance with the information that it needs to play musical patterns. For an overview, see Using Styles.

Because styles usually include bands and motifs, the **IDirectMusicStyle8** interface provides methods for accessing these objects.

The DirectMusicStyle object also supports the **IDirectMusicObject8** and **IPersistStream** interfaces for loading its data.

IDirectMusicStyle8 supersedes the **IDirectMusicStyle** interface and provides a new method, **EnumPattern**.

The methods of the **IDirectMusicStyle8** interface can be organized in the following groups:

Enumeration	**EnumBand**
	EnumChordMap
	EnumMotif
	EnumPattern
Information	**GetBand**
	GetChordMap
	GetDefaultBand
	GetDefaultChordMap
	GetEmbellishmentLength
	GetMotif
	GetTempo
	GetTimeSignature

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusici.h.

IDirectMusicStyle8::EnumBand

Retrieves the name of the band with a given index value.

```
HRESULT EnumBand(
  DWORD dwIndex,
  WCHAR * pwszName
);
```

Parameters

dwIndex
Zero-based index into the style's band list.

pwszName
Address of a buffer that receives the band name. This should be of size MAX_PATH.

Return Values

If the method succeeds, it returns S_OK, S_FALSE if there is no band with the given index value, or DMUS_S_STRING_TRUNCATED if the length of the name is greater than MAX_PATH.

If it fails, the method can return one of the following error values:

DMUS_E_TYPE_UNSUPPORTED

E_POINTER

Requirements

Header: Declared in dmusici.h.

IDirectMusicStyle8::EnumChordMap

Retrieves the name of the chordmap with the given index value.

```
HRESULT EnumChordMap(
  DWORD dwIndex,
  WCHAR * pwszName
);
```

Parameters

dwIndex
Zero-based index of the chordmap in the style's chordmap list.

pwszName
Address of a buffer that receives the chordmap name. This should be of size MAX_PATH.

Return Values

If the method succeeds, the return value is S_OK, S_FALSE if there is no chordmap with the given index value, or DMUS_S_STRING_TRUNCATED if the length of the name is greater than MAX_PATH.

If it fails, the method can return one of the following error values:

DMUS_E_TYPE_UNSUPPORTED
E_POINTER

Requirements

Header: Declared in dmusici.h.

IDirectMusicStyle8::EnumMotif

Retrieves the name of a motif with a given index value.

```
HRESULT EnumMotif(
  DWORD dwIndex,
  WCHAR* pwszName
);
```

Parameters

dwIndex
Zero-based index into the style's motif list.

pwszName
Address of a buffer that receives the motif name. This should be of size MAX_PATH.

Return Values

If the method succeeds, the return value is S_OK, S_FALSE if there is no motif with the given index value, or DMUS_S_STRING_TRUNCATED if the length of the motif name is greater than MAX_PATH.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle8::GetMotif, Using Motifs

IDirectMusicStyle8::EnumPattern

Retrieves the name of a pattern with a given index value and type. The name can be passed to the **IDirectMusicPatternTrack8::SetPatternByName** method.

```
HRESULT EnumPattern(
  DWORD dwIndex,
```

```
  DWORD dwPatternType,
  WCHAR* pwszName
);
```

Parameters

dwIndex
Zero-based index into the style's pattern list.

dwPatternType
One of the **DMUS_STYLET_TYPES** enumeration that specifies the type of pattern.

pwszName
Address of a buffer that receives the pattern name. This should be of size MAX_PATH.

Return Values

If the method succeeds, the return value is S_OK, S_FALSE if there is no pattern with the given index value and type, or DMUS_S_STRING_TRUNCATED if the length of the motif name is greater than MAX_PATH.

If it fails, the method can return one of the following error values.

E_INVALIDARG

E_POINTER.

Requirements

Header: Declared in dmusici.h.

IDirectMusicStyle8::GetBand

Retrieves the named band.

```
HRESULT GetBand(
  WCHAR* pwszName,
  IDirectMusicBand** ppBand
);
```

Parameters

pwszName
Name of the band to be retrieved. This name is assigned by the author of the style.

ppBand

Address of a variable that receives the **IDirectMusicBand8** interface pointer of the band.

Return Values

If the method succeeds, the return value is S_OK if a band is returned, or S_FALSE if there is no band with that name.

If the method fails, the return value can be E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle8::GetDefaultBand, Using Bands

IDirectMusicStyle8::GetChordMap

Retrieves a named chordmap.

```
HRESULT GetChordMap(
  WCHAR* pwszName,
  IDirectMusicChordMap** ppChordMap
);
```

Parameters

pwszName
Name of the chordmap to be retrieved.

ppChordMap
Address of a variable that receives a pointer to the **IDirectMusicChordMap8** interface.

Return Values

If the method succeeds, the return value is S_OK if a chordmap is returned, or S_FALSE if there is no chordmap by that name.

If *ppChordMap* is not a valid pointer, the method returns E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle8::GetDefaultChordMap, Using Chordmaps

IDirectMusicStyle8::GetDefaultBand

Retrieves the style's default band.

```
HRESULT GetDefaultBand(
  IDirectMusicBand ** ppBand
);
```

Parameters

ppBand
Address of a variable that receives the **IDirectMusicBand8** interface pointer for the default band.

Return Values

If the method succeeds, the return value is S_OK if a band is returned, or S_FALSE if the style does not have a default band.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle8::GetBand, Using Bands

IDirectMusicStyle8::GetDefaultChordMap

Retrieves the style's default chordmap.

```
HRESULT GetDefaultChordMap(
  IDirectMusicChordMap** ppChordMap
);
```

Parameters

ppChordMap
Address of a variable that receives a pointer to the **IDirectMusicChordMap8** interface.

Return Values

If the method succeeds, the return value is S_OK if a chordmap is returned, or S_FALSE if the style does not have a default chordmap.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle8::GetChordMap, Using Chordmaps

IDirectMusicStyle8::GetEmbellishmentLength

Finds the shortest and longest lengths for patterns of the specified embellishment type and groove level.

```
HRESULT GetEmbellishmentLength(
  DWORD dwType,
  DWORD dwLevel,
  DWORD* pdwMin,
  DWORD* pdwMax
);
```

Parameters

dwType
Embellishment type. See **DMUS_COMMANDT_TYPES**.

dwLevel
Groove level, in the range from 1 through 100. Ignored if *dwType* is not DMUS_COMMANDT_GROOVE.

pdwMin
Address of a variable that receives the length, in measures, of the shortest pattern of the specified type and groove level.

pdwMax
Address of a variable that receives the length, in measures, of the longest pattern of the specified type and groove level.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE.

If it fails, the method can return E_POINTER.

Remarks

If there are no patterns of the specified type and groove level, the method returns S_FALSE.

Requirements

Header: Declared in dmusici.h.

IDirectMusicStyle8::GetMotif

Creates a segment containing the named motif.

```
HRESULT GetMotif(
  WCHAR* pwszName,
  IDirectMusicSegment** ppSegment
);
```

Parameters

pwszName
Name of the motif to be retrieved.

ppSegment
Address of a variable that receives a pointer to a segment containing the named motif.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE.

If it fails, the method can return E_POINTER.

Remarks

The method searches the style's list of motifs for one whose name matches *pwszName*. If one is found, a segment is created containing a motif track. The track references the style as its associated style and the motif as its pattern.

If there is no motif with the name, the method returns S_FALSE.

Requirements

Header: Declared in dmusici.h.

See Also

Using Motifs

IDirectMusicStyle8::GetTempo

Retrieves the recommended tempo of the style.

```
HRESULT GetTempo(
  double* pTempo
);
```

Parameters

pTempo
Address of a variable that receives the recommended tempo of the style.

Return Values

If the method succeeds, the return value is S_OK.

If *pTempo* is not a valid pointer, the method returns E_POINTER.

Requirements

Header: Declared in dmusici.h.

IDirectMusicStyle8::GetTimeSignature

Retrieves the style's time signature.

```
HRESULT GetTimeSignature(
  DMUS_TIMESIGNATURE* pTimeSig
);
```

Parameters

pTimeSig
Address of a **DMUS_TIMESIGNATURE** structure that receives data.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmusici.h.

IDirectMusicThru8

The **IDirectMusicThru8** interface supports thruing of music messages from a capture port to another port. It is obtained by calling **QueryInterface** on the **IDirectMusicPort8** interface for the capture port. For an example, see the Remarks for **IDirectMusicThru8::ThruChannel**.

IDirectMusicThru8 is a type definition for **IDirectMusicThru**. The two interface names can be used interchangeably.

The interface has the following method:

IDirectMusicThru8	**ThruChannel**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTMUSICTHRU8** type is defined as a pointer to this interface.

```
typedef IDirectMusicThru8 *LPDIRECTMUSICTHRU8;
```

Requirements

Header: Declared in dmusici.h.

See Also

Capturing MIDI

IDirectMusicThru8::ThruChannel

Establishes or breaks a thruing connection between a channel on a capture port and a channel on another port.

```
HRESULT ThruChannel(
  DWORD dwSourceChannelGroup,
  DWORD dwSourceChannel,
  DWORD dwDestinationChannelGroup,
  DWORD dwDestinationChannel,
  LPDIRECTMUSICPORT pDestinationPort
);
```

Parameters

dwSourceChannelGroup

Channel group on the capture port. This value is always 1.

dwSourceChannel
Source channel.

dwDestinationChannelGroup
Channel group on the destination port.

dwDestinationChannel
Destination channel.

pDestinationPort
Address of the **IDirectMusicPort8** interface for the destination channel. Set this value to NULL to break an existing thruing connection.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_NOTIMPL

E_INVALIDARG

DMUS_E_PORT_NOT_RENDER

Remarks

System exclusive messages are not transmitted to the destination port.

Thruing to the Microsoft software synthesizer or other synthesizers that do not have a constant latency is not recommended. Thruing is done as soon as possible upon reception of the incoming MIDI events. Because of the comparatively high latency of the software synthesizer (compared with a hardware port) and the fact that it renders blocks of audio data at the same time, each event is delayed by a small, essentially random amount of time before it plays. This random offset shows up as jitter in the playback of the data. Latency of other devices (such as an MPU-401 port) is small enough that jitter does not occur.

If an application needs to thru to the software synthesizer, it should add a small offset to the incoming note event time stamps to compensate for the rendering latency of the synthesizer.

The following code example obtains the **IDirectMusicThru8** interface and establishes a thru connection between all channels on group 1 of the capture port and the equivalent channels on a destination port.

```
HRESULT SetupOneToOneThru(
    IDirectMusicPort8 *pCapturePort,
    IDirectMusicPort8 *pRenderPort)
{
    HRESULT hr;
    IDirectMusicThru8 *pThru;
```

```
    hr = pCapturePort->QueryInterface(IID_IDirectMusicThru8,
          (void**)&pThru);
    if (FAILED(hr))
       return hr;

    for (DWORD dwChannel = 0; dwChannel < 16; dwChannel++)
    {
       hr = pThru->ThruChannel(1, dwChannel,
          1, dwChannel, (IDirectMusicPort*)pRenderPort);
       if (FAILED(hr))
          break;
    }

    pThru->Release();
    return hr;
}
```

Requirements

Header: Declared in dmusici.h.

IDirectMusicTool8

The **IDirectMusicTool8** interface represents a tool object that processes messages. The tool can modify a message, create additional messages, remove messages, and so on.

This interface is of interest chiefly to developers who want to create their own tools. Methods of the interface are implemented by the designer of the tool and are generally called by the performance. The application only needs to insert the tool in the message path by using **IDirectMusicGraph8::InsertTool**.

IDirectMusicTool8 supersedes the **IDirectMusicTool** interface and adds a new method.

The methods of the **IDirectMusicTool8** interface can be organized in the following groups:

Duplication	**Clone**
Initialization	**Init**
Message management	**Flush**
	GetMediaTypeArraySize
	GetMediaTypes
	GetMsgDeliveryType
	ProcessPMsg

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmplugin.h.

See Also

Message Creation and Delivery, DirectMusic Tools

IDirectMusicTool8::Clone

Creates a new instance of the tool.

```
HRESULT Clone(
  IDirectMusicTool** ppTool
);
```

Parameters

ppTool

Address of a variable that receives a pointer to the **IDirectMusicTool** interface of the new instance of the tool. Use QueryInterface to obtain **IDirectMusicTool8**.

Return Values

Return values are determined by the implementation. If it succeeds, the method should return S_OK. If it fails, the return value might be E_POINTER or E_OUTOFMEMORY.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTool8::Flush

Flushes messages from the queue when the performance stops. The tool can use the method to do whatever is necessary to flush the message. For instance, the output tool uses this method to ensure that any pending note-off messages are processed immediately.

```
HRESULT Flush(
  IDirectMusicPerformance* pPerf,
  DMUS_PMSG* pPMSG,
  REFERENCE_TIME rtTime
);
```

Parameters

pPerf
Address of the **IDirectMusicPerformance8** interface.

pPMSG
Message to flush.

rtTime
Time at which to flush.

Return Values

Return values are determined by the implementation. If the method succeeds, the return value can be one of the following:

DMUS_S_REQUEUE

DMUS_S_FREE

S_OK

If it fails, the method can return E_POINTER.

Remarks

The message will have DMUS_PMSGF_TOOL_FLUSH set in its **dwFlags** member. See **DMUS_PMSG**.

If the method returns DMUS_S_REQUEUE, the tool wants the message to be requeued. This allows the tool to put a new time stamp and parameters on the message and requeue it, or to requeue the message with a different delivery type.

If the return value is DMUS_S_FREE, the tool wants the message freed automatically and does not want to requeue the message.

If S_OK is returned, the tool does not want the message to be freed automatically. Perhaps the tool is holding onto the message for some reason, or has freed it itself.

Be sure not to create a circular reference to the performance represented by *pPerf*. For more information, see DirectMusic Tools.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTool8::GetMediaTypeArraySize

Retrieves the size of the array that must be passed in to the **IDirectMusicTool8::GetMediaTypes** method. A return value of 0 indicates that the tool handles all types, and it is unnecessary to call **GetMediaTypes**.

```
HRESULT GetMediaTypeArraySize(
  DWORD* pdwNumElements
);
```

Parameters

pdwNumElements
Address of a variable that receives the number of media types. If 0 is returned in this field, all types are supported.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK. If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTool8::GetMediaTypes

Retrieves a list of the types of messages that this tool supports.

```
HRESULT GetMediaTypes(
  DWORD** padwMediaTypes,
  DWORD dwNumElements
);
```

Parameters

padwMediaTypes
Address of an array of **DWORD**s. The method fills this array with the media types supported by this tool. For media types, see **DMUS_PMSGT_TYPES**.

dwNumElements
Number of elements in the *padwMediaTypes* array. This value is equal to the number returned by the **IDirectMusicTool8::GetMediaTypeArraySize** method. If *dwNumElements* is less than this number, the method cannot return all the message types that are supported. If it is greater than this number, the extra elements in the array should be set to 0.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK, or S_FALSE if the method could not fill in all values because *dwNumElements* was too small. If it fails, the method can return one of the following error values:

E_POINTER

E_INVALIDARG

E_NOTIMPL

Remarks

If the method returns E_NOTIMPL, the tool processes all media types.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTool8::GetMsgDeliveryType

Retrieves the tool's delivery type, which determines when messages are to be delivered to the tool.

```
HRESULT GetMsgDeliveryType(
  DWORD* pdwDeliveryType
);
```

Parameters

pdwDeliveryType

Address of a variable that receives the delivery type. The returned value must be DMUS_PMSGF_TOOL_IMMEDIATE, DMUS_PMSGF_TOOL_QUEUE, or DMUS_PMSGF_TOOL_ATTIME. An unrecognized value in **pdwDeliveryType* is treated as DMUS_PMSGF_TOOL_IMMEDIATE by the graph.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK. If it fails, the method can return E_POINTER.

Remarks

For an overview of the delivery mechanism, see Message Creation and Delivery.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTool8::Init

Initializes the tool. This method is called when the tool is inserted into the graph, giving the tool an opportunity to perform any necessary initialization.

```
HRESULT Init(
  IDirectMusicGraph* pGraph
);
```

Parameters

pGraph
Calling graph.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK. If it fails, the method can return one of the following error values:

E_FAIL
E_NOTIMPL

Remarks

Because a tool can be inserted into more than one graph, this method must be able to deal gracefully with multiple calls.

Be sure not to create a circular reference to the graph represented by *pGraph*. For more information, see DirectMusic Tools.

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicGraph8::InsertTool

IDirectMusicTool8::ProcessPMsg

Performs the main task of the tool. It is called from inside the performance's real-time thread for all messages that match the types specified by **IDirectMusicTool8::GetMediaTypes**.

```
HRESULT ProcessPMsg(
  IDirectMusicPerformance* pPerf,
  DMUS_PMSG* pPMSG
);
```

Parameters

pPerf
Performance that is generating messages.

pPMSG
Message to process.

Return Values

Return values are determined by the implementation. If the method succeeds, the return value can be one of the following:

DMUS_S_REQUEUE

DMUS_S_FREE

S_OK

If it fails, the method can return E_POINTER.

Remarks

If the method returns DMUS_S_REQUEUE, the tool wants the message to be requeued. This allows the tool to put a new time stamp and parameters on the message and requeue it, or to requeue the message with a different delivery type.

If the return value is DMUS_S_FREE, the tool wants the message freed automatically, and does not want to requeue the message.

If S_OK is returned, the tool does not want the message to be freed automatically. Perhaps the tool is holding onto the message for some reason, or has freed it itself.

Tools should not perform time-consuming activities because doing so can severely affect overall performance. Also be sure not to create a circular reference to the performance represented by *pPerf*. For more information, see DirectMusic Tools.

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicPerformance8::SendPMsg, Messages, Message Creation and Delivery

IDirectMusicTrack8

The **IDirectMusicTrack8** interface represents a track object. Almost everything that has to do with the definition of a segment is stored in its tracks. The track mechanism allows segments to be infinitely extensible, and the segment does not need any knowledge of any of the music and audio technologies that it employs.

If you plan to install your own music playback mechanism in DirectMusic, you need to create a DirectMusicTrack object to represent it. Otherwise, the methods of this interface are typically not called directly from the application.

IDirectMusicTrack8 supersedes the **IDirectMusicTrack** interface and adds new methods.

Note

When implementing methods of the **IDirectMusicTrack8** interface, be sure not to hold onto references to objects passed in. For example, if **IDirectMusicTrack8::Init** adds a reference to the **IDirectMusicSegment** interface that it receives as a parameter, ensure that this reference is released.

The **IDirectMusicTrack8** interface has the following methods:

Creation	**Clone**
	Compose
	Join
Initialization	**Init**
Notification	**AddNotificationType**
	RemoveNotificationType
Parameters	**GetParam**
	GetParamEx
	IsParamSupported
	SetParam
	SetParamEx
Playback	**EndPlay**
	InitPlay
	Play
	PlayEx

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The DirectMusicTrack object also supports the **IDirectMusicObject8** and **IPersistStream** interfaces for loading its data.

Requirements

Header: Declared in dmplugin.h.

See Also

DirectMusic Tracks, Setting and Retrieving Track Parameters

IDirectMusicTrack8::AddNotificationType

Enables event notification for a track. It is similar to and called from the **IDirectMusicSegment8::AddNotificationType** method.

```
HRESULT AddNotificationType(
  REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType
Reference to (C++) or address of (C) the identifier of the notification type to add. For the defined types, see **DMUS_NOTIFICATION_PMSG**. Applications can also define their own types for custom tracks.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the track does not support the notification type.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT

E_NOTIMPL

Remarks

If the track does not support notifications, the method returns E_NOTIMPL. A motif or style track returns DMUS_E_NOT_INIT if it has not been initialized.

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicTrack8::RemoveNotificationType, Notification and Event Handling

IDirectMusicTrack8::Clone

Makes a copy of a track.

```
HRESULT Clone(
  MUSIC_TIME mtStart,
  MUSIC_TIME mtEnd,
  IDirectMusicTrack** ppTrack
);
```

Parameters

mtStart
Start of the part to copy. It should be 0 or greater and less than the length of the track.

mtEnd
End of the part to copy. It should be greater than *mtStart* and less than the length of the track.

ppTrack
Address of a variable that receives a pointer to the created track, if successful.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG
E_OUTOFMEMORY
E_POINTER

Remarks

It is the caller's responsibility to call **Release** when finished with the track.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTrack8::Compose

Recomposes the track based on data from a segment. DirectMusic implements this method on the signpost track to compose a chord track.

```
HRESULT Compose(
  IUnknown* pContext,
  DWORD dwTrackGroup,
  IDirectMusicTrack** ppResultTrack
);
```

Parameters

pContext
: **IUnknown** interface pointer of the object to use in the composition. This is usually the segment that owns this track.

dwTrackGroup
: **DWORD** value that specifies group bits for the track. For more information on group bits, see Identifying the Track.

ppResultTrack
: Address of a variable that receives the **IDirectMusicTrack** interface of the composed track. Use **QueryInterface** to obtain **IDirectMusicTrack8**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

E_INVALIDARG

E_POINTER

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicSegment8::Compose

IDirectMusicTrack8::EndPlay

Called when the object that originally called **IDirectMusicTrack8::InitPlay** is destroyed.

```
HRESULT EndPlay(
  void * pStateData
```

```
);
```

Parameters

pStateData
Pointer to state data returned from **IDirectMusicTrack8::InitPlay**. This data should be freed in the **EndPlay** method.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTrack8::GetParam

Retrieves data from a track, in music time.

```
HRESULT GetParam(
  REFGUID rguidType,
  MUSIC_TIME mtTime,
  MUSIC_TIME* pmtNext,
  void* pParam
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data to obtain. See Standard Track Parameters.

mtTime
Time, in track time, from which to obtain the data.

pmtNext
Address of a variable that receives the track time (relative to the current time) until which the data is valid. If this returns a value of 0, either the data is always valid, or it is unknown when it might become invalid. If this information is not needed, *pmtNext* can be set to NULL.

pParam
Address of an allocated structure in which the data is to be returned. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
DMUS_E_NOT_INIT
DMUS_E_TYPE_DISABLED
DMUS_E_GET_UNSUPPORTED
E_POINTER

Remarks

The **IDirectMusicTrack8::GetParamEx** method can be used for greater functionality.

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicTrack8::SetParam, **IDirectMusicTrack8::IsParamSupported**, **IDirectMusicPerformance8::GetParam**, **IDirectMusicSegment8::GetParam**, Performance Parameters

IDirectMusicTrack8::GetParamEx

Retrieves data from a track, in either music or reference time.

```
HRESULT GetParamEx(
  REFGUID rguidType,
  REFERENCE_TIME rtTime,
  REFERENCE_TIME* prtNext,
  void * pParam
  void * pStateData,
  DWORD dwFlags
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data to obtain. See Standard Track Parameters.

rtTime

Time from which to obtain the data. Unless DMUS_TRACK_PARAMF_CLOCK is set in *dwFlags*, this value is in music time.

prtNext
Address of a variable that receives the time until which the data is valid. If this returns a value of zero, either the data is always valid, or it is unknown when it might become invalid. If this information is not needed, *prtNext* can be set to NULL.

pParam
Address of an allocated structure in which the data is to be returned. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

pStateData
Address of a buffer containing state data for the track instance. This value is obtained from **IDirectMusicTrack8::InitPlay**.

dwFlags
Can be zero or the following flag.

DMUS_TRACK_PARAMF_CLOCK
The value in *rtTime* is in clock time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

DMUS_E_NOT_FOUND
DMUS_E_NOT_INIT
DMUS_E_TYPE_DISABLED
DMUS_E_GET_UNSUPPORTED
DMUS_E_TRACK_NO_CLOCKTIME_SUPPORT
E_POINTER

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicTrack8::GetParam, **IDirectMusicTrack8::SetParamEx**

IDirectMusicTrack8::Init

Initializes the track. This method is called by a segment when a track is added.

```
HRESULT Init(
  IDirectMusicSegment* pSegment
);
```

Parameters

pSegment
Segment to which this track belongs.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_OUTOFMEMORY
E_POINTER

Remarks

If the track plays messages, it should call **IDirectMusicSegment8::SetPChannelsUsed** in the **Init** method.

This method should be called whenever track data is changed after the track is inserted in a segment.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTrack8::InitPlay

Called when a track is ready to start playing. The method returns a pointer to state data.

```
HRESULT InitPlay(
  IDirectMusicSegmentState* pSegmentState,
  IDirectMusicPerformance* pPerformance,
  void** ppStateData,
  DWORD dwVirtualTrackID,
  DWORD dwFlags
);
```

Parameters

pSegmentState

Address of the calling **IDirectMusicSegmentState** or **IDirectMusicSegmentState8** interface.

pPerformance
Address of the calling **IDirectMusicPerformance** or **IDirectMusicPerformance8** interface.

ppStateData
Address of a variable that receives a pointer to state information. The format and use of the data is specific to the track. The data should be created in the **InitPlay** method and freed in the **IDirectMusicTrack8::EndPlay** method. The pointer is passed to the **IDirectMusicTrack8::Play** and **IDirectMusicTrack8::PlayEx** methods.

dwVirtualTrackID
Virtual track ID assigned to this track instance.

dwFlags
DMUS_SEGF_FLAGS that control the track's behavior. See Remarks.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT

E_OUTOFMEMORY

E_POINTER

Remarks

The *dwFlags* parameter passes the flags that were handed to the performance in the call to **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx**. The track determines how it should perform, based on the DMUS_SEGF_CONTROL and DMUS_SEGF_SECONDARY flags. For example, the tempo track automatically plays the tempo changes only if it is part of a primary segment or a secondary control segment (DMUS_SEGF_SECONDARY is not set, or DMUS_SEGF_CONTROL is set).

A track can return NULL in *ppStateData*.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTrack8::IsParamSupported

Determines whether the track supports a given data type in the **IDirectMusicTrack8::GetParam** and **IDirectMusicTrack8::SetParam** methods.

```
HRESULT IsParamSupported(
  REFGUID rguidType
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data. See Standard Track Parameters.

Return Values

If the method succeeds and the type is supported, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_TYPE_DISABLED

DMUS_E_TYPE_UNSUPPORTED

E_POINTER

E_NOTIMPL

Remarks

If a parameter has been disabled by using one of the **SetParam** methods, the **IDirectMusicTrack8::IsParamSupported** method returns DMUS_E_TYPE_DISABLED when passed the corresponding parameter type (either GUID_TempoParam or GUID_TimeSignature).

The method also returns DMUS_E_TYPE_DISABLED if passed GUID_DisableTempo when that parameter has already been disabled, or if passed GUID_EnableTempo when that parameter is currently enabled. The same is true for GUID_DisableTimeSig and GUID_EnableTimeSig.

The method returns DMUS_E_TYPE_UNSUPPORTED when the track does not support the parameter referred to by a GUID_EnableTempo, GUID_EnableTimeSig, GUID_DisableTempo, or GUID_DisableTimeSig parameter call.

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicTrack8::GetParam, **IDirectMusicTrack8::SetParam**, Performance Parameters

IDirectMusicTrack8::Join

Appends one track to another.

```
HRESULT Join(
  IDirectMusicTrack* pNewTrack,
  MUSIC_TIME mtJoin,
  IUnknown* pContext,
  DWORD dwTrackGroup,
  IDirectMusicTrack** ppResultTrack)
);
```

Parameters

pNewTrack
Pointer to an **IDirectMusicTrack8** interface that specifies the track to append to this one.

mtJoin
Time within this track where *pNewTrack* is to begin.

pContext
IUnknown interface pointer of the context segment. This object determines the time signature for tracks that use measures and beats, such as the signpost track.

dwTrackGroup
Group or groups to which the new track belongs. For more information on track groups, see **IDirectMusicSegment8::InsertTrack** and Identifying the Track.

ppResultTrack
Address of a variable that receives the **IDirectMusicTrack** interface of the concatenated track. Use **QueryInterface** to obtain **IDirectMusicTrack8**. If NULL, no new track is created and the current track becomes the concatenated track.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

E_POINTER
E_INVALIDARG
E_OUTOFMEMORY

Remarks

This method is supported by the band, chordmap, tempo, style, chord, signpost, and command tracks.

Requirements

Header: Declared in dmplugin.h.

IDirectMusicTrack8::Play

Causes the track to play. It performs any work that the track must do when the segment is played, such as creating and sending messages.

```
HRESULT Play(
  void* pStateData,
  MUSIC_TIME mtStart,
  MUSIC_TIME mtEnd,
  MUSIC_TIME mtOffset
  DWORD dwFlags,
  IDirectMusicPerformance* pPerf,
  IDirectMusicSegmentState* pSegSt,
  DWORD dwVirtualID
);
```

Parameters

pStateData
Pointer to state data from the **IDirectMusicTrack8::InitPlay** method. The format and use of the data is specific to the track.

mtStart
Start time.

mtEnd
End time.

mtOffset
Offset to add to all messages sent to **IDirectMusicPerformance8::SendPMsg**.

dwFlags
Flags that indicate the state of this call. See **DMUS_TRACKF_FLAGS**. A value of 0 indicates that this call to **Play** continues playback from the previous call.

pPerf
Performance used to allocate and send messages.

pSegSt
Segment state that this track belongs to. The **IDirectMusicSegmentState8::QueryInterface** method can be called to obtain an **IDirectMusicGraph8** interface—to call **IDirectMusicGraph8::StampPMsg**, for instance.

dwVirtualID
Virtual identifier of the track. This value must be put in the **dwVirtualTrackID** member of any message (see **DMUS_PMSG**) that is sent by **IDirectMusicPerformance8::SendPMsg**.

Return Values

If the method succeeds, the return value can be S_OK or DMUS_S_END.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_POINTER

Remarks

If the track is empty, the method returns DMUS_S_END.

Tracks generate messages in a medium-priority thread. You can call time-consuming functions, such as code to stream data from a file, from within a track's **Play** method. However, be sure to follow the guidelines for safe multithreading.

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicTrack8::PlayEx

IDirectMusicTrack8::PlayEx

Causes the track to play in clock time. It performs any work that the track must do when the segment is played, such as creating and sending messages.

```
HRESULT PlayEx(
  void* pStateData,
  REFERENCE_TIME rtStart,
  REFERENCE_TIME rtEnd,
  REFERENCE_TIME rtOffset,
  DWORD dwFlags,
  IDirectMusicPerformance* pPerf,
  IDirectMusicSegmentState* pSegSt,
  DWORD dwVirtualID
);
```

Parameters

pStateData
Pointer to state data from the **IDirectMusicTrack8::InitPlay** method. The format and use of the data is specific to the track.

rtStart
Start time.

rtEnd
End time.

rtOffset
Offset to add to all messages sent to **IDirectMusicPerformance8::SendPMsg**.

dwFlags
Flags that indicate the state of this call. See **DMUS_TRACKF_FLAGS**. A value of zero indicates that this call to **PlayEx** continues playback from the previous call.

pPerf
Performance used to allocate and send messages.

pSegSt
Segment state that this track belongs to. The **IDirectMusicSegmentState8::QueryInterface** method can be called to obtain an **IDirectMusicGraph8** interface—for instance, to call **IDirectMusicGraph8::StampPMsg**.

dwVirtualID
Virtual identifier of the track. This value must be put in the **dwVirtualTrackID** member of any message (see **DMUS_PMSG**) that is sent by **IDirectMusicPerformance8::SendPMsg**.

Return Values

If the method succeeds, the return value can be S_OK or DMUS_S_END.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_POINTER

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicTrack8::Play

IDirectMusicTrack8::RemoveNotificationType

Removes an event notification from a track. It is similar to and called from the **IDirectMusicSegment8::RemoveNotificationType** method.

HRESULT RemoveNotificationType(

```
  REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType
Reference to (C++) or address of (C) the identifier of the notification type to remove. For the defined types, see **DMUS_NOTIFICATION_PMSG**.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the track does not support the notification type.

If the track does not support notifications, the method returns E_NOTIMPL.

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicTrack8::AddNotificationType, Notification and Event Handling

IDirectMusicTrack8::SetParam

Sets data on a track, in music time.

```
HRESULT SetParam(
  REFGUID rguidType,
  MUSIC_TIME mtTime,
  void* pParam
);
```

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data to set. See Standard Track Parameters.

mtTime
Time, in track time, at which to set the data.

pParam
Address of a structure containing the data, or NULL if no data is required. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_SET_UNSUPPORTED
DMUS_E_TYPE_DISABLED
E_OUTOFMEMORY
E_POINTER

Remarks

The **IDirectMusicTrack8::SetParamEx** method can be used for greater functionality.

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicTrack8::GetParam, **IDirectMusicTrack8::IsParamSupported**, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::SetParam**, Setting and Retrieving Track Parameters

IDirectMusicTrack8::SetParamEx

Sets data on a track, in either clock or music time.

```
HRESULT SetParamEx(
  REFGUID rguidType,
  REFERENCE_TIME rtTime,
  void* pParam,
  void * pStateData,
  DWORD dwFlags
 );
```

Parameters

rguidType
Reference to (C++) or address of (C) the identifier of the type of data to set. See Standard Track Parameters.

rtTime
Time at which to set the data. Unless DMUS_TRACK_PARAMF_CLOCK is set in dwFlags, this is in music time.

pParam

Address of a structure that contains the data, or NULL if no data is required. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

pStateData
Pointer to a buffer that contains state data for the track.

dwFlags
Can be zero or the following flag.

DMUS_TRACK_PARAMF_CLOCK
The value in *rtTime* is in clock time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values.

DMUS_E_SET_UNSUPPORTED
DMUS_E_TYPE_DISABLED
E_OUTOFMEMORY
E_POINTER

Requirements

Header: Declared in dmplugin.h.

See Also

IDirectMusicTrack8::GetParamEx, **IDirectMusicTrack8::SetParam**

IKsControl

The **IKsControl** interface is used to get, set, or query the support of properties, events, and methods. This interface is part of the Windows Driver Model kernel streaming architecture, but is also used by DirectMusic to expose properties of DirectMusic ports. To retrieve this interface, call the **IDirectMusicPort8::QueryInterface** method with IID_IKsControl in the *riid* parameter.

Routing of the property item request to the port varies, depending on the port implementation. No properties are supported by ports that represent DirectMusic emulation on top of the Win32 handle-based multimedia calls (**midiOut** and **midiIn** functions).

Property item requests to a port that represents a pluggable software synthesizer are answered totally in user mode. The topology of this type of port is a synthesizer (represented by an **IDirectMusicSynth** interface) connected to a sink node (an

IDirectMusicSynthSink interface). The property request is given first to the synthesizer node, and then to the sink node if it is not recognized by the synthesizer.

The interface has the following methods. At present, only **KsProperty** is supported by DirectMusic.

IKsControl	**KsProperty**
	KsEvent
	KsMethod

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmksctrl.h.

See Also

Property Sets for DirectMusic Ports

IKsControl::KsProperty

Retrieves or sets the value of a property. For an overview, see Property Sets for DirectMusic Ports.

```
HRESULT KsProperty(
  PKSPROPERTY pProperty,
  ULONG ulPropertyLength,
  LPVOID pvPropertyData,
  ULONG ulDataLength,
  PULONG pulBytesReturned
);
```

Parameters

pProperty

Address of a **KSPROPERTY** structure that gives the property set, item, and operation to perform. If this property contains instance data, that data should reside in memory immediately following the structure.

ulPropertyLength

Length of the memory pointed to by *pProperty*, including any instance data.

pvPropertyData

For a set operation, the address of a memory buffer containing data that represents the new value of the property. For a get operation, the address of a memory buffer big enough to hold the value of the property. For a basic support query, the address of a buffer at least a **DWORD** in size.

ulDataLength
Length of the buffer pointed to by *pvPropertyData*.

pulBytesReturned
On a KSPROPERTY_TYPE_GET or KSPROPERTY_TYPE_BASICSUPPORT call, address of a variable that receives the number of bytes returned in *pvPropertyData* by the port.

Return Values

If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG
E_NOTIMPL
E_OUTOFMEMORY
E_POINTER
DMUS_E_UNKNOWN_PROPERTY

Requirements

Header: Declared in dmksctrl.h.

See Also

Property Sets for DirectMusic Ports

IReferenceClock

The **IReferenceClock** interface represents a system reference clock. The DirectMusic master clock and a port's latency clock implement this interface.

The interface has the following methods:

IReferenceClock	**GetTime**
	AdviseTime
	AdvisePeriodic
	Unadvise

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dmusicc.h.

See Also

IDirectMusic8::GetMasterClock, **IDirectMusicPort8::GetLatencyClock**, Timing

IReferenceClock::AdvisePeriodic

Requests an asynchronous, periodic notification that a duration has elapsed.

```
HRESULT AdvisePeriodic(
  REFERENCE_TIME rtStartTime,
  REFERENCE_TIME rtPeriodTime,
  HANDLE hSemaphore,
  DWORD * pdwAdviseCookie
);
```

Parameters

rtStartTime
Time that the notification should begin.

rtPeriodTime
Period of time between notifications.

hSemaphore
Handle of a semaphore through which to advise.

pdwAdviseCookie
Address of a variable that receives the identifier of the request. This is used to identify this call to **AdvisePeriodic** in the future—for example, to cancel it.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_FAIL

E_POINTER

E_INVALIDARG

E_NOTIMPL

Remarks

When the time indicated by *rtStartTime* is reached, the semaphore whose handle is set as *hSemaphore* is released. Thereafter, the semaphore is released repetitively with a period of *rtPeriodTime*.

Requirements

Header: Declared in dmusicc.h.

See Also

IReferenceClock::Unadvise

IReferenceClock::AdviseTime

Requests an asynchronous notification that a time has elapsed.

```
HRESULT AdviseTime(
  REFERENCE_TIME rtBaseTime,
  REFERENCE_TIME rtStreamTime,
  HANDLE hEvent,
  DWORD * pdwAdviseCookie
);
```

Parameters

rtBaseTime
Base reference time.

rtStreamTime
Stream offset time.

hEvent
Handle to an event through which to advise.

pdwAdviseCookie
Address of a variable that receives the identifier of the request. This is used to identify this call to **AdviseTime** in the future—for example, to cancel it.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_FAIL

E_POINTER

E_INVALIDARG
E_NOTIMPL

Remarks

When the time *rtBaseTime* plus *rtStreamTime* is reached, the event whose handle is *hEvent* is set. If the time has already passed, the event is set immediately.

Requirements

Header: Declared in dmusicc.h.

See Also

IReferenceClock::Unadvise

IReferenceClock::GetTime

Retrieves the current time.

```
HRESULT GetTime(
  REFERENCE_TIME * pTime
);
```

Parameters

pTime
Address of a variable that receives the current time.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_POINTER
E_INVALIDARG
E_NOTIMPL

Requirements

Header: Declared in dmusicc.h.

IReferenceClock::Unadvise

Cancels a request for notification.

```
HRESULT Unadvise(
  DWORD dwAdviseCookie
);
```

Parameters

dwAdviseCookie
Identifier of the request that is to be canceled, as set in the **IReferenceClock::AdviseTime** or the **IReferenceClock::AdvisePeriodic** method.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_POINTER
E_INVALIDARG
E_NOTIMPL

Requirements

Header: Declared in dmusicc.h.

DirectMusic Messages

DirectMusic message structures are all based on the **DMUS_PMSG** structure. Because C does not support inheritance, the members of this structure are included in each derived structure as the **DMUS_PMSG_PART** macro.

For an overview of messages, see Using DirectMusic Messages.

This section contains information about the following structures used to contain message information:

- **DMUS_PMSG**

- **DMUS_CHANNEL_PRIORITY_PMSG**
- **DMUS_CURVE_PMSG**
- **DMUS_LYRIC_PMSG**
- **DMUS_MIDI_PMSG**
- **DMUS_NOTE_PMSG**
- **DMUS_NOTIFICATION_PMSG**
- **DMUS_PATCH_PMSG**
- **DMUS_SYSEX_PMSG**
- **DMUS_TEMPO_PMSG**
- **DMUS_TIMESIG_PMSG**
- **DMUS_TRANSPOSE_PMSG**
- **DMUS_WAVE_PMSG**

See Also

IDirectMusicPerformance8::AllocPMsg, **IDirectMusicPerformance8::SendPMsg**, **IDirectMusicPerformance8::FreePMsg**, **IDirectMusicTool8::ProcessPMsg**

DMUS_PMSG

Contains information common to all DirectMusic messages. Because C does not support inheritance, the members of this structure are contained in all message types (including **DMUS_PMSG** itself) as the **DMUS_PMSG_PART** macro, which expands to the syntax shown here.

```
typedef struct DMUS_PMSG {
    DWORD             dwSize;
    REFERENCE_TIME    rtTime;
    MUSIC_TIME        mtTime;
    DWORD             dwFlags;
    DWORD             dwPChannel;
    DWORD             dwVirtualTrackID;
    IDirectMusicTool* pTool;
    IDirectMusicGraph* pGraph;
    DWORD             dwType;
    DWORD             dwVoiceID;
    DWORD             dwGroupID;
    IUnknown*         punkUser;
} DMUS_PMSG;
```

dwSize

Size of the structure, in bytes. This member is initialized by **IDirectMusicPerformance8::AllocPMsg**.

rtTime
Reference time at which the message is to be played, modified by **dwFlags**. Used only if DMUS_PMSGF_REFTIME is present in **dwFlags**.

mtTime
Music time at which the message is to be played, modified by **dwFlags**. Used only if DMUS_PMSGF_MUSICTIME is present in **dwFlags**.

dwFlags
Flags from the **DMUS_PMSGF_FLAGS** or **DMUS_TIME_RESOLVE_FLAGS** enumeration. Must contain DMUS_PMSGF_REFTIME or DMUS_PMSGF_MUSICTIME.

dwPChannel
Performance channel (PChannel). The port, channel group, and MIDI channel can be derived from this value by using the **IDirectMusicPerformance8::PChannelInfo** method. Set this value to 0 for messages that are not channel-specific, such as tempo messages. To send the message to more than channel, use one of the following values.

DMUS_PCHANNEL_BROADCAST_AUDIOPATH
Send a copy of the message to all channels of the audiopath.

DMUS_PCHANNEL_BROADCAST_GROUPS
Send a copy of the message to each channel group in the performance. Used for messages that need to be sent only once per channel group, such as system exclusive messages.

DMUS_PCHANNEL_BROADCAST_PERFORMANCE
Send a copy of the message to all channels of the performance.

DMUS_PCHANNEL_BROADCAST_SEGMENT
Send a copy of the message to all channels of the segment.

dwVirtualTrackID
Identifier of the track. Set to 0 if the message is not being sent by a track.

pTool
Address of the tool interface. Can be set by using **IDirectMusicGraph8::StampPMsg**, or can be NULL if the message is not to go to tools other than the output tool.

pGraph
Address of the tool graph interface. Can be set by using **IDirectMusicGraph8::StampPMsg**, or can be NULL if the message is not to go to tools other than the output tool.

dwType
Message type (see **DMUS_PMSGT_TYPES**).

dwVoiceID
Reserved. Must be 0.

dwGroupID
Identifier of the track group or groups that the message belongs to if the message is being generated by a track. (Tracks are assigned to groups in the

IDirectMusicSegment8::InsertTrack method.) For most purposes, this value can be 0xFFFFFFF.

punkUser
Address of an **IUnknown** interface supplied by the application. This pointer is always released when the message is freed. If the application wants to retain the object, it should call **AddRef** before the message is freed. If the message does not need a COM pointer, this value should be NULL.

Remarks

The **DMUS_PMSG** structure is used by itself for messages containing the following values in the **dwType** member:

DMUS_PMSGT_STOP
Sending a message of this type stops the performance at the specified time.

DMUS_PMSGT_DIRTY
When a control segment starts or ends, all tools in the segment and performance graphs receive a message of this type, indicating that if they cache data from get-parameter calls, they must call the method again to refresh their data. Tools that want to receive this message type must indicate this through a call to **IDirectMusicTool8::GetMediaTypes**. Tools in the performance graph receive one copy of the message for each segment in the performance. Such tools can safely ignore the extra messages with the same time stamp.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SendPMsg

DMUS_CHANNEL_PRIORITY_PMSG

Contains message data about a channel priority change.

```
typedef struct _DMUS_CHANNEL_PRIORITY_PMSG {
    DMUS_PMSG_PART
    DWORD dwChannelPriority;
} DMUS_CHANNEL_PRIORITY_PMSG;
```

DMUS_PMSG_PART
Macro for common message members. See **DMUS_PMSG**.

dwChannelPriority

Priority of the channel. For a list of defined values, see the Remarks for **IDirectMusicPort8::GetChannelPriority**.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPort8::SetChannelPriority, **IDirectMusicPerformance8::SendPMsg**

DMUS_CURVE_PMSG

Contains message data for a curve.

```
typedef struct DMUS_CURVE_PMSG {
    DMUS_PMSG_PART
    MUSIC_TIME mtDuration;
    MUSIC_TIME mtOriginalStart;
    MUSIC_TIME mtResetDuration;
    short      nStartValue;
    short      nEndValue;
    short      nResetValue;
    WORD       wMeasure;
    short      nOffset;
    BYTE       bBeat;
    BYTE       bGrid;
    BYTE       bType;
    BYTE       bCurveShape;
    BYTE       bCCData;
    BYTE       bFlags;
    WORD       wParamType;
    WORD       wMergeIndex;
} DMUS_CURVE_PMSG;
```

DMUS_PMSG_PART
Macro for common message members. See **DMUS_PMSG**.

mtDuration
Duration of the curve. This value is in music time unless DMUS_PMSGF_LOCKTOREFTIME is present in the **dwFlags** member of **DMUS_PMSG_PART**, in which case it is in milliseconds and is unaffected by a change in tempo.

mtOriginalStart
Original start time. Must be set to either 0 when this message is created, or to the original start time of the curve.

mtResetDuration

Length of time after the end of the curve during which a reset can take place in response to an invalidation. Ignored if DMUS_CURVE_RESET is not in **bFlags**. This value is in music time unless DMUS_PMSGF_LOCKTOREFTIME is present in the **dwFlags** member of **DMUS_PMSG_PART**, in which case it is in milliseconds and is unaffected by a change in tempo.

nStartValue
Start value of the curve.

nEndValue
End value of the curve.

nResetValue
Value to set upon a flush or invalidation. Ignored if DMUS_CURVE_RESET is not in **bFlags**.

wMeasure
Measure in which this curve occurs.

nOffset
Offset from the grid at which this curve occurs, in music time.

bBeat
Beat count (within a measure) at which this curve occurs.

bGrid
Grid offset from the beat at which this curve occurs.

bType
Type of curve. This can be one of the following values:

DMUS_CURVET_CCCURVE
Continuous controller curve (MIDI Control Change channel voice message; status byte &HB*n*.).

DMUS_CURVET_MATCURVE
Monophonic aftertouch curve (MIDI Channel Pressure channel voice message; status byte &HD*n*).

DMUS_CURVET_PATCURVE
Polyphonic aftertouch curve (MIDI Poly Key Pressure channel voice message, status byte &HD*n*).

DMUS_CURVET_PBCURVE
Pitch-bend curve (MIDI Pitch Bend channel voice message; status byte &HE*n*).

DMUS_CURVET_RPNCURVE
RPN curve of type defined in **wParamType**.

DMUS_CURVET_NRPNCURVE
NRPN curve of type defined in **wParamType**.

bCurveShape
Shape of curve. This can be one of the following values:

DMUS_CURVES_EXP
Exponential curve shape.

DMUS_CURVES_INSTANT
Instant curve shape (beginning and end of curve happen at essentially the same time).

DMUS_CURVES_LINEAR
Linear curve shape.

DMUS_CURVES_LOG
Logarithmic curve shape.

DMUS_CURVES_SINE
Sine curve shape.

bCCData
Controller number, if **bType** is DMUS_CURVET_CCCURVE.

bFlags
Can be zero, or one or more of the following values.

DMUS_CURVE_RESET
The value of **nResetValue** must be set when the time is reached or an invalidation occurs because of a transition. If this flag is not set, the curve stays permanently at the new value.

DMUS_CURVE_START_FROM_CURRENT
Ignore **nStartValue** and start the curve at the current value. Implemented for volume, expression, pitch bend, filter cuttoff, pan, and mod wheel. See Remarks.

wParamType
MIDI parameter number. This value is significant only if DMUS_PMSGF_DX8 is present in the **dwFlags** member of the **DMUS_PMSG** part of this structure. See Remarks.

wMergeIndex
Merge index. Supported for pitch bend, volume, and expression controllers. This value is significant only if DMUS_PMSGF_DX8 is present in the **dwFlags** member of the **DMUS_PMSG** part of this structure. See Remarks.

Remarks

An RPN or NRPN curve type in **wParamType** is stored as two bytes with seven significant bits. For example, if the MSB is 0x23 and the LSB is 0x74, the value in **wParamType** is 0x2374.

Data in **nStartValue**, **nEndValue**, and **nResetValue** is limited to 14 bits. For MIDI data consisting of two seven-bit bytes, the value is stored as a word with the upper two bits empty.

All curves with **wMergeIndex** of 0 override each other. If **wMergeIndex** is another value, the values generated by the curve are added to the values for merge index 0. For example, if an application uses curves with 0 and 3, the 0 curves always replace each other but add to the 3 curves, and the 3 curves also always replace each other and add to the 0 curves.

The DMUS_CURVE_START_FROM_CURRENT flag does not cause the current controller value to be saved in the message. Therefore tools do not have access to this data unless they store the last known value.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SendPMsg, Curves

DMUS_LYRIC_PMSG

Contains message data for a string.

```
typedef struct _DMUS_LYRIC_PMSG {
    DMUS_PMSG_PART
    WCHAR  wszString[1];
} DMUS_LYRIC_PMSG;
```

DMUS_PMSG_PART
Macro for common message members. See **DMUS_PMSG**.

wszString
Null-terminated Unicode string. The array is sized when the message is created.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SendPMsg

DMUS_MIDI_PMSG

Contains data for a standard MIDI message such as a control change or pitch bend.

```
typedef struct DMUS_MIDI_PMSGG {
    DMUS_PMSG_PART
    BYTE  bStatus;
    BYTE  bByte1;
    BYTE  bByte2;
    BYTE  bPad[1];
} DMUS_MIDI_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

bStatus
Standard MIDI status byte.

bByte1
First byte of the MIDI message. Ignored for MIDI messages that do not require it.

bByte2
Second byte of the MIDI message. Ignored for MIDI messages that do not require it.

bPad
Padding to a **WORD** boundary.

Requirements

Header: Declared in dmusici.h.

See Also

MIDI Messages, **IDirectMusicPerformance8::SendPMsg**

DMUS_NOTE_PMSG

Contains message data for a MIDI note.

```
typedef struct DMUS_NOTE_PMSG {
    DMUS_PMSG_PART
    MUSIC_TIME mtDuration;
    WORD       wMusicValue;
    WORD       wMeasure;
    short      nOffset;
    BYTE       bBeat;
    BYTE       bGrid;
    BYTE       bVelocity;
    BYTE       bFlags;
    BYTE       bTimeRange;
    BYTE       bDurRange;
    BYTE       bVelRange;
    BYTE       bPlayModeFlags;
    BYTE       bSubChordLevel;
    BYTE       bMidiValue;
    char       cTranspose;
} DMUS_NOTE_PMSG;
```

DMUS_PMSG_PART
Macro for common message members. See **DMUS_PMSG**.

mtDuration
Duration of the note.

wMusicValue
Description of the note. In most play modes, this is a packed array of 4-bit values, as follows:

Octave
In the range from –2 through 14. The note is transposed up or down by the octave times 12.

Chord position
In the range from 0 through 15, although it should never be above 3. The first position in the chord is 0.

Scale position
In the range from 0 through 15. Typically it is only from 0 through 2, but it is possible to have a one-note chord and have everything above the chord be interpreted as a scale position.

Accidental
In the range from –8 through 7, but typically in the range from –2 through 2. This represents an offset that takes the note out of the scale.

In the fixed-play modes, the music value is a MIDI note value in the range from 0 through 127.

wMeasure
Measure in which this note occurs.

nOffset
Offset from the grid at which this note occurs, in music time.

bBeat
Beat (in measure) at which this note occurs.

bGrid
Grid offset from the beat at which this note occurs.

bVelocity
Note velocity.

bFlags
See **DMUS_NOTEF_FLAGS**.

bTimeRange
Range by which to randomize time.

bDurRange
Range by which to randomize duration.

bVelRange
Range by which to randomize velocity.

bPlayModeFlags
Play mode determining how the music value is related to the chord and subchord. For a list of values, see **DMUS_PLAYMODE_FLAGS**.

bSubChordLevel
Subchord level that the note uses. See **DMUS_SUBCHORD**.

bMidiValue
MIDI note value, converted from **wMusicValue**.

cTranspose
Transposition to add to **bMidiValue** after conversion from **wMusicValue**.

Remarks

When the output tool receives a message with DMUS_NOTEF_NOTEON in **bFlags**, it sends a MIDI note-on message to the synthesizer. It then clears the DMUS_NOTEF_NOTEON flag, adds **mtDuration** to the time stamp, and requeues the message so that the note is turned off at the appropriate time.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SendPMsg, Music Values and MIDI Notes

DMUS_NOTIFICATION_PMSG

Contains message data for a notification.

```
typedef struct DMUS_NOTIFICATION_PMSG {
    DMUS_PMSG_PART
    GUID   guidNotificationType;
    DWORD  dwNotificationOption;
    DWORD  dwField1;
    DWORD  dwField2;
} DMUS_NOTIFICATION_PMSG;
```

DMUS_PMSG_PART
Macro for common message members. See **DMUS_PMSG**.

guidNotificationType
Identifier of the notification type. The following types are defined:

GUID_NOTIFICATION_CHORD
Chord change.

GUID_NOTIFICATION_COMMAND
Command event.

GUID_NOTIFICATION_MEASUREANDBEAT
Measure and beat event.

GUID_NOTIFICATION_PERFORMANCE
Performance event, further defined in **dwNotificationOption**.

GUID_NOTIFICATION_RECOMPOSE

A track has been recomposed. In the current version, this is always a chord track.

GUID_NOTIFICATION_SEGMENT
Segment event, further defined in **dwNotificationOption**.

dwNotificationOption

Identifier of the notification subtype.

If the notification type is GUID_NOTIFICATION_SEGMENT, this member can contain one of the following values:

DMUS_NOTIFICATION_SEGABORT
The segment was stopped prematurely, or was removed from the primary segment queue.

DMUS_NOTIFICATION_SEGALMOSTEND
The segment has reached the end minus the prepare time.

DMUS_NOTIFICATION_SEGEND
The segment has ended.

DMUS_NOTIFICATION_SEGLOOP
The segment has looped.

DMUS_NOTIFICATION_SEGSTART
The segment has started.

If the notification type is GUID_NOTIFICATION_COMMAND, this member can contain one of the following values:

DMUS_NOTIFICATION_GROOVE
Groove level change.

DMUS_NOTIFICATION_EMBELLISHMENT
Embellishment command (intro, fill, break, or end).

If the notification type is GUID_NOTIFICATION_PERFORMANCE, this member can contain one of the following values:

DMUS_NOTIFICATION_MUSICALMOSTEND
The currently playing primary segment has reached the end minus the prepare time, and no more primary segments are cued to play.

DMUS_NOTIFICATION_MUSICSTARTED
Playback has started.

DMUS_NOTIFICATION_MUSICSTOPPED
Playback has stopped.

If the notification type is GUID_NOTIFICATION_MEASUREANDBEAT, this member contains DMUS_NOTIFICATION_MEASUREBEAT. No other subtypes are defined.

If the notification type is GUID_NOTIFICATION_CHORD, this member contains DMUS_NOTIFICATION_CHORD. No other subtypes are defined.

If the notification type is GUID_NOTIFICATION_RECOMPOSE, this member contains DMUS_NOTIFICATION_RECOMPOSE. No other subtypes are defined.

dwField1
Extra data specific to the type of notification. For GUID_NOTIFICATION_MEASUREANDBEAT notifications, this member returns the beat number within the measure.

dwField2
Extra data specific to the type of notification. Reserved for future or application-defined use.

Remarks

For most notifications, the **punkUser** member (see **DMUS_PMSG**) contains the **IUnknown** pointer of the segment state. This is especially useful in the cases of chords and commands, in which you can query for the **IDirectMusicSegmentState8** interface, call **IDirectMusicSegmentState8::GetSegment** to get the **IDirectMusicSegment8** pointer, and then call the **IDirectMusicSegment8::GetParam** method to get the chord or command at the time given in the notification message's **mtTime** member.

For notifications of type GUID_NOTIFICATION_PERFORMANCE, the **punkUser** member is always NULL.

Applications can define their own notification message types and subtypes and use **dwField1** and **dwField2** for extra data. Such custom notification messages can be allocated and sent like any other message. Application-defined tracks can send messages of a particular type after the GUID **(guidNotificationType)** has been handed to **IDirectMusicTrack8::AddNotificationType**.

Requirements

Header: Declared in dmusici.h.

See Also

Notification and Event Handling, **IDirectMusicPerformance8::SendPMsg**

DMUS_PATCH_PMSG

Contains message data for a MIDI program change.

```
typedef struct DMUS_PATCH_PMSG {
    DMUS_PMSG_PART
    BYTE  byInstrument;
    BYTE  byMSB;
    BYTE  byLSB;
```

```
    BYTE  byPad[1];
} DMUS_PATCH_PMSG;
```

DMUS_PMSG_PART
Macro for common message members. See **DMUS_PMSG**.

byInstrument
Patch number of the instrument.

byMSB
Most significant byte of bank select.

byLSB
Least significant byte of bank select.

byPad
Padding to a **WORD** boundary. This value is ignored.

Requirements

Header: Declared in dmusici.h.

See Also

DMUS_MIDI_PMSG, **IDirectMusicPerformance8::SendPMsg**, MIDI Messages

DMUS_SYSEX_PMSG

Contains data for a MIDI system exclusive message.

```
typedef struct DMUS_SYSEX_PMSG {
    DMUS_PMSG_PART
    DWORD dwLen;
    BYTE  abData[1];
} DMUS_SYSEX_PMSG;
```

DMUS_PMSG_PART
Macro for common message members. See **DMUS_PMSG**.

dwLen
Length of the data, in bytes.

abData
Array of data. For an example of how to allocate memory and copy data to this member, see the Remarks for **IDirectMusicPerformance8::SendPMsg**.

Remarks

The data part of a system exclusive message must begin with the System Exclusive identifier (0xF0) and end with EOX (0xF7).

Requirements

Header: Declared in dmusici.h.

See Also

DMUS_MIDI_PMSG, **DMUS_PATCH_PMSG**, MIDI Messages, **IDirectMusicPerformance8::SendPMsg**

DMUS_TEMPO_PMSG

Contains data for a message that controls the performance's tempo.

```
typedef struct DMUS_TEMPO_PMSG {
    DMUS_PMSG_PART
    double dblTempo;
} DMUS_TEMPO_PMSG;
```

DMUS_PMSG_PART
Macro for common message members. See **DMUS_PMSG**.

dblTempo
Tempo, in the range from DMUS_TEMPO_MIN through DMUS_TEMPO_MAX.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SendPMsg

DMUS_TIMESIG_PMSG

Contains data for a message that controls the time signature of the performance.

```
typedef struct _DMUS_TIMESIG_PMSG {
    DMUS_PMSG_PART
    BYTE bBeatsPerMeasure;
    BYTE bBeat;
    WORD wGridsPerBeat;
} DMUS_TIMESIG_PMSG;
```

DMUS_PMSG_PART
Macro for common message members. See **DMUS_PMSG**.

bBeatsPerMeasure

Beats per measure (top of the time signature).

bBeat

Note that receives the beat (bottom of the time signature), where 1 is a whole note, 2 is a half note, 4 is a quarter note, and so on. Zero is a 256th note.

wGridsPerBeat

Grids (subdivisions) per beat. This value determines the timing resolution for certain music events—for example, segments cued with the DMUS_SEGF_GRID flag (see **DMUS_SEGF_FLAGS**).

Remarks

Time signature messages are generated by the time signature track and the style track. In general, a segment contains one or the other, but not both. A segment representing a MIDI file has a time signature track, but most segments authored with an application such as DirectMusic Producer contain time signature information in the style track.

By default, only the primary segment sends time signature messages. For information on how to change this behavior, see Disabling and Enabling Track Parameters.

The time signature is used by the performance to resolve time to measure, beat, and grid boundaries in all methods in which the time can be adjusted by **DMUS_SEGF_FLAGS** or **DMUS_TIME_RESOLVE_FLAGS**. The time signature and style tracks also use the time signature to generate notifications on measure and beat boundaries. See **DMUS_NOTIFICATION_PMSG**.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SendPMsg, **DMUS_TIMESIGNATURE**

DMUS_TRANSPOSE_PMSG

Contains message data for a transposition.

```
typedef struct _DMUS_TRANSPOSE_PMSG {
    DMUS_PMSG_PART
    short nTranspose;
    WORD wMergeIndex;
} DMUS_TRANSPOSE_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

nTranspose

Number of semitones by which to transpose. This can be a negative value.

wMergeIndex

Merge index. When a transpose message follows a preceding message with the same **wMergeIndex**, the value in **nTranspose** becomes the new setting. When the second transpose message has a different **wMergeIndex**, the value in **nTranspose** is added to the previous setting. This member is significant only if DMUS_PMSGF_DX8 is present in the **dwFlags** member of the **DMUS_PMSG** part of this structure.

Remarks

If the transposition of a note puts it outside the standard MIDI range from 0 through 127, it does not play.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SendPMsg

DMUS_WAVE_PMSG

Contains message data for a wave sound.

```
typedef struct _DMUS_WAVE_PMSG {
    DMUS_PMSG_PART
    REFERENCE_TIME rtStartOffset;
    REFERENCE_TIME rtDuration;
    long    lOffset;
    long    lVolume;
    long    lPitch;
    BYTE    bFlags;
} DMUS_WAVE_PMSG;
;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**. The **punkUser** member contains the address of the **IUnknown** interface of the voice object associated with the wave.

rtStartOffset

How far into the wave to start, in reference time units only.

rtDuration

Duration of the wave. If DMUS_PMSGF_LOCKTOREFTIME is present in the **dwFlags** member of **DMUS_PMSG_PART**, this value is in reference time units. Otherwise it is in music time.

lOffset
Offset from actual time to logical time, in either reference or music time.

lVolume
Initial volume, in hundredths of a decibel.

lPitch
Transposition of the pitch, in hundredths of a semitone.

bFlags
Can be zero or the following value.

DMUS_WAVEF_NOINVALIDATE
Do not invalidate this wave.

DMUS_WAVEF_OFF
This message is stopping playback of the wave.

DMUS_WAVEF_STREAMING
Wave is streaming.

Remarks

Applications cannot send messages of this type by using **IDirectMusicPerformance8::SendPMsg**, because they have no way of obtaining a pointer to a wave object. However, tools can process wave messages.

Requirements

Header: Declared in dmusici.h.

DirectMusic Structures

This section contains reference information for the following run-time structures used in DirectMusic:

- **DMUS_AUDIOPARAMS**
- **DMUS_BAND_PARAM**
- **DMUS_BUFFERDESC**
- **DMUS_CHORD_KEY**
- **DMUS_CHORD_PARAM**
- **DMUS_CLOCKINFO8**
- **DMUS_COMMAND_PARAM**
- **DMUS_COMMAND_PARAM_2**

- **DMUS_CONNECTION_RULE**
- **DMUS_EVENTHEADER**
- **DMUS_MUTE_PARAM**
- **DMUS_NOTERANGE**
- **DMUS_OBJECTDESC**
- **DMUS_PLAY_MARKER_PARAM**
- **DMUS_PORTCAPS**
- **DMUS_PORTPARAMS8**
- **DMUS_RHYTHM_PARAM**
- **DMUS_SCRIPT_ERRORINFO**
- **DMUS_SUBCHORD**
- **DMUS_SYNTHSTATS8**
- **DMUS_TEMPO_PARAM**
- **DMUS_TIMESIGNATURE**
- **DMUS_VALID_START_PARAM**
- **DMUS_VARIATIONS_PARAM**
- **DMUS_VERSION**
- **DMUS_WAVES_REVERB_PARAMS**
- **KSPROPERTY**

Special categories of structures are contained in the following sections:

- DirectMusic Messages
- DirectMusic File Structures
- DLS Structures

DMUS_AUDIOPARAMS

Describes required resources for the default synthesizer and buffers of a performance. Passed to the **IDirectMusicPerformance8::InitAudio** method to request desired features and to receive information about what requests were granted.

```
typedef struct _DMUS_AUDIOPARAMS {
    DWORD  dwSize;
    BOOL   fInitNow;
    DWORD  dwValidData;
    DWORD  dwFeatures;
    DWORD  dwVoices;
    DWORD  dwSampleRate;
    CLSID  clsidDefaultSynth;
} DMUS_AUDIOPARAMS;
```

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

fInitNow
Boolean value that specifies whether the sink and synthesizer are created immediately. If so, results are returned in this structure.

dwValidData
Flags that specify which members of this structure are valid. If **fInitNow** is TRUE when the structure is passed, this member receives flags that specify what members received data. The following values are defined.

DMUS_AUDIOPARAMS_FEATURES
The **dwFeatures** member contains or has received data.

DMUS_AUDIOPARAMS_VOICES
The **dwVoices** member contains or has received data.

DMUS_AUDIOPARAMS_SAMPLERATE
The **dwSampleRate** member contains or has received data.

DMUS_AUDIOPARAMS_DEFAULTSYNTH
The **clsidDefaultSynth** member contains or has received data. If this flag is not set, the Microsoft software synthesizer is the default synthesizer.

dwFeatures
Flags that specify required capabilities. The following values are defined.

DMUS_AUDIOF_3D
3-D buffers.

DMUS_AUDIOF_ALL
Combination of all other flags.

DMUS_AUDIOF_BUFFERS
Multiple buffers. Always set this flag when using audiopaths.

DMUS_AUDIOF_ENVIRON
Environmental modeling.

DMUS_AUDIOF_EAX
EAX effects.

DMUS_AUDIOF_DMOS
Additional DMOs.

DMUS_AUDIOF_STREAMING
Support for streaming waves.

dwVoices
Number of voices. The default value is 64.

dwSampleRate
Sample rate of sink and synthesizer, in the range from 11,025 to 96,000 kHz. The default value is 22,050.

clsidDefaultSynth

Class identifier of the default synthesizer. This is the synthesizer used by standard audiopaths and audiopaths created from configurations that request the default synthesizer.

Requirements

Header: Declared in dmusicf.h.

DMUS_BAND_PARAM

Used as the *pParam* parameter in calls to the various get-parameter and set-parameter methods when the track is a band track and *rguidType* is GUID_BandParam.

```
typedef struct _DMUS_BAND_PARAM {
    MUSIC_TIME mtTimePhysical;
    IDirectMusicBand *pBand;
} DMUS_BAND_PARAM;
```

Members

mtTimePhysical
Actual time at which the band change will be made. See Remarks.

pBand
Address of the **IDirectMusicBand8** interface of the band.

Remarks

The value in **mtTimePhysical** is the actual time at which the band change will be made, whereas the value in the *mtTime* parameter of the set-parameter method is the point in the performance where the change belongs, for example, synchronized with a beat or measure. You can set **mtTimePhysical** to a time slightly before *mtTime* to ensure that notes are always played by the correct band, even when a band change is made at the start of a loop.

If the track is a clock-time track, *mtTimePhysical* is interpreted in the track's internal time format. This is the number of milliseconds after the beginning of playback. Because this can be confusing, it is recommended that GUID_BandParam not be used with clock-time tracks.

Requirements

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::GetParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicTrack8::SetParamEx**, Performance Parameters

DMUS_BUFFERDESC

Used to describe a buffer for the **IDirectMusic8::CreateMusicBuffer** method.

```
typedef struct _DMUS_BUFFERDESC {
    DWORD dwSize;
    DWORD dwFlags;
    GUID  guidBufferFormat;
    DWORD cbBuffer;
} DMUS_BUFFERDESC, *LPDMUS_BUFFERDESC;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

dwFlags
No flags are defined.

guidBufferFormat
Identifier of the KS format of the buffer. The value GUID_NULL represents KSDATAFORMAT_SUBTYPE_DIRECTMUSIC.

If **guidBufferFormat** represents a KS format other than KSDATAFORMAT_SUBTYPE_DIRECTMUSIC, the application must verify that the port playing back the data understands the specified format; if not, the buffer is ignored. To find out whether the port supports a specific KS format, use the **IKsControl::KsProperty** method.

cbBuffer
Minimum size of the buffer, in bytes. The amount of memory allocated can be slightly higher because the system pads the buffer to a multiple of 4 bytes. The buffer must be at least 32 bytes to accommodate a single MIDI channel message, and at least 28 bytes plus the size of the data to accommodate a system exclusive message or other unstructured data.

Requirements

Header: Declared in dmusici.h.

See Also

DMUS_EVENTHEADER, **IDirectMusicBuffer8::PackStructured**, **IDirectMusicBuffer8::PackUnstructured**

DMUS_CHORD_KEY

Used to describe a chord in the **IDirectMusicPerformance8::MIDIToMusic** and **IDirectMusicPerformance8::MusicToMIDI** methods.

```
typedef struct _DMUS_CHORD_KEY {
    WCHAR        wszName[16];
    WORD         wMeasure;
    BYTE         bBeat;
    BYTE         bSubChordCount;
    DMUS_SUBCHORD SubChordList[DMUS_MAXSUBCHORD];
    DWORD        dwScale;
    BYTE         bKey;
    BYTE         bFlags;
} DMUS_CHORD_KEY;
```

Members

wszName
Name of the chord.

wMeasure
Measure that the chord falls on.

bBeat
Beat that the chord falls on.

bSubChordCount
Number of chords in the chord's list of subchords.

SubChordList
Array of **DMUS_SUBCHORD** structures, describing the components that make up the chord.

dwScale
Scale underlying the entire chord.

bKey
Key underlying the entire chord.

bFlags
Can be zero, or DMUS_CHORDKEYF_SILENT if the chord is silent. See Remarks.

Remarks

This structure is also defined as a **DMUS_CHORD_PARAM** structure for use in setting and retrieving the GUID_ChordParam track parameter.

If a chord is flagged as a silent chord, it is not taken into consideration when a pattern is selected to be played. For instance, if there is a chord change on beat 1 and the silent chord is on beat 3, a pattern with a whole measure chord rhythm can still be played.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::GetParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicTrack8::SetParamEx**, Performance Parameters

DMUS_CHORD_PARAM

Used as the *pParam* parameter in calls to the various get-parameter and set-parameter methods when the track is a chord track and *rguidType* is GUID_ChordParam.

```
typedef DMUS_CHORD_KEY DMUS_CHORD_PARAM;
```

See **DMUS_CHORD_KEY**.

Requirements

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::GetParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicTrack8::SetParamEx**, Performance Parameters

DMUS_CLOCKINFO8

Reports information about a clock enumerated by using the **IDirectMusic8::EnumMasterClock** method.

```
typedef struct _DMUS_CLOCKINFO{
    DWORD          dwSize;
```

```
    DMUS_CLOCKTYPE  ctType;
    GUID            guidClock;
    WCHAR           wszDescription[DMUS_MAX_DESCRIPTION];
    DWORD           dwFlags;
} DMUS_CLOCKINFO8, *LPDMUS_CLOCKINFO8;

typedef DMUS_CLOCKINFO8 DMUS_CLOCKINFO;
typedef DMUS_CLOCKINFO *LPDMUS_CLOCKINFO;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is passed to a method.

ctType
Member of the **DMUS_CLOCKTYPE** enumeration specifying the type of clock.

guidClock
Identifier of the clock. This value can be passed to the **IDirectMusic8::SetMasterClock** method to set the master clock for DirectMusic.

wszDescription
Description of the clock.

dwFlags
Flags.

Requirements

Header: Declared in dmusici.h.

DMUS_COMMAND_PARAM

Used as the *pParam* parameter in calls to various get-parameter and set-parameter methods when the track is a command track and the *rguidType* parameter is GUID_CommandParam.

```
typedef struct {
    BYTE bCommand;
    BYTE bGrooveLevel;
    BYTE bGrooveRange;
    BYTE bRepeatMode;
} DMUS_COMMAND_PARAM;
```

Members

bCommand
Command type. See **DMUS_COMMANDT_TYPES**.

bGrooveLevel
Groove level of the command. The groove level is a value in the range from 1 through 100.

bGrooveRange
Amount by which the groove level can be randomized. For instance, if the groove level is 35 and the range is 4, the actual groove level could be anywhere from 33 through 37. If **bGrooveRange** is an odd number, 1 is subtracted from it.

bRepeatMode
Flag that specifies how patterns are selected for repetition. See **DMUS_PATTERNT_TYPES**.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::GetParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicTrack8::SetParamEx**

DMUS_COMMAND_PARAM_2

Used as the *pParam* parameter in calls to various get-parameter and set-parameter methods when the track is a command track and the *rguidType* parameter is GUID_CommandParam2.

```
typedef struct _DMUS_COMMAND_PARAM_2 {
    MUSIC_TIME mtTime;
    BYTE bCommand;
    BYTE bGrooveLevel;
    BYTE bGrooveRange;
    BYTE bRepeatMode;
} DMUS_COMMAND_PARAM_2;
```

Members

mtTime
Time of the command.

bCommand
Command type. See **DMUS_COMMANDT_TYPES**.

bGrooveLevel
Groove level of the command. The groove level is a value in the range from 1 through 100.

bGrooveRange
Amount by which the groove level can be randomized. For instance, if the groove level is 35 and the range is 4, the groove level could be anywhere from 33 through 37. If **bGrooveRange** is an odd number, 1 is subtracted from it.

bRepeatMode
Flag that specifies how patterns are selected for repetition. See **DMUS_PATTERNT_TYPES**.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::GetParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicTrack8::SetParamEx**

DMUS_CONNECTION_RULE

Not implemented in DirectX 8.0.

Used in the **ConnectionArc** member of the **DMUS_MELODY_FRAGMENT** to specify rules for allowing fragments to flow smoothly from one to another.

```
typedef struct _DMUS_CONNECTION_RULE {
    DWORD      dwFlags;
    DWORD      dwIntervals;
} DMUS_CONNECTION_RULE;
```

Members

dwFlags
Flags that specify the connection rules. Can contain zero or more of the following values.

DMUS_CONNECTIONF_INTERVALS
Use the intervals in **dwIntervals** between the last note of one variation and the first note of the next.

DMUS_CONNECTIONF_OVERLAP
If the two fragments overlap, the first note not played in the first fragment is the same as the first note in the second fragment.

dwIntervals

If DMUS_CONNECTIONF_INTERVALS is set, specifies a set of legal intervals between the last note played in the first fragment and the first note in the second fragment. Each of the lower 24 bits is an interval in a two-octave range.

Requirements

Header: Declared in dmusici.h.

DMUS_EVENTHEADER

Precedes and describes an event in a port buffer.

```
typedef struct _DMUS_EVENTHEADER {
    DWORD          cbEvent;
    DWORD          dwChannelGroup;
    REFERENCE_TIME  rtDelta;
    DWORD          dwFlags;
} DMUS_EVENTHEADER, *LPDMUS_EVENTHEADER;
```

Members

cbEvent
Number of bytes in the event.

dwChannelGroup
Group to which the event belongs.

rtDelta
Offset from the start time of the buffer.

dwFlags
Set to DMUS_EVENT_STRUCTURED if the event is parsable MIDI data.

Remarks

The Pshpack4.h header file is included before the declaration of this structure to turn off automatic alignment of structures so that the data immediately follows the header. (For more information, see the comments in Pshpack4.h.) Poppack.h is then included to turn alignment back on, and the entire structure (header plus event) is padded to an 8-byte boundary.

Requirements

Header: Declared in dmusbuff.h.

See Also

IDirectMusicBuffer8::GetNextEvent, **IDirectMusicBuffer8::PackStructured**, **IDirectMusicBuffer8::PackUnstructured**

DMUS_MELODY_FRAGMENT

Not implemented in DirectX 8.0.

Describes a fragment in a melody formulation track. A melody fragment is used to select a variation from a pattern during melody composition.

```
typedef struct _DMUS_MELODY_FRAGMENT {
    MUSIC_TIME  mtTime;
    DWORD  dwID;
    WCHAR  wszVariationLabel[DMUS_MAX_FRAGMENTLABEL];
    DWORD  dwVariationFlags;
    DWORD  dwRepeatFragmentID;
    DWORD  dwFragmentFlags;
    DWORD  dwPlayModeFlags;
    DWORD  dwTransposeIntervals;
    DMUS_COMMAND_PARAM   Command;
    DMUS_CONNECTION_RULE ConnectionArc;
} DMUS_MELODY_FRAGMENT;
```

Members

mtTime
Time at which the fragment is to play, relative to the beginning of the track.

dwID
Identifier of the track.

wszVariationLabel
Reserved.

dwVariationFlags
DWORD value that specifies the set of allowable variations, where each bit set represents one variation. Ignored if DMUS_FRAGMENTF_USE_REPEAT is included in **dwFragmentFlags**.

dwRepeatFragmentID
Identifier of an earlier fragment in the track from which to select a variation. Ignored unless DMUS_FRAGMENTF_USE_REPEAT is included in **dwFragmentFlags**.

dwFragmentFlags
Can contain any of the following values.

DMUS_FRAGMENTF_USE_REPEAT
Use the fragment specified in **dwRepeatFragmentID**.

DMUS_FRAGMENTF_REJECT_REPEAT

Use a new variation of the earlier fragment.

DMUS_FRAGMENTF_USE_LABEL
Use the set named by **wszVariationLabel**. Not supported.

dwPlayModeFlags
Reserved. Must be 0.

dwTransposeIntervals
The lower 24 bits each represent an interval in a two-octave range. Specifies a set of intervals by which the notes in a repeated fragment can be transposed to fit the current chord or key. The lower 24 bits each represent an interval in a two-octave range.

Command
DMUS_COMMAND_PARAM structure that specifies the type of pattern to use as the source for the melody.

ConnectionArc
DMUS_CONNECTION_RULE structure that specifies rules for allowing fragments to flow smoothly from one to another.

Remarks

The variations selected for each fragment form a melody. Fragments can be placed on any beat boundary and can cross bar boundaries. If two fragments overlap, the variation selected by the first fragment is truncated so that none of its notes run into the notes from the second fragment.

Each melody fragment contains a description of the variations that are suitable at that point. This could be a range of variations, a list, or some designation for "use all variations." The variation selection can also refer to previous variations. A melody fragment can use a play mode to generate notes in the melody being constructed, or it can select variations that align with the current chord using either chord intervals or scale intervals. Further, a set of intervals can be selected as legal intervals by which to transpose the fragment in order to align it.

Requirements

Header: Declared in dmusici.h.

See Also

GUID_MelodyFragment

DMUS_MUTE_PARAM

Used as the *pParam* parameter in calls to the various get-parameter and set-parameter methods when the track is a mute track and *rguidType* is GUID_MuteParam.

```
typedef struct _DMUS_MUTE_PARAM {
```

```
    DWORD  dwPChannel;
    DWORD  dwPChannelMap;
    BOOL   fMute;
} DMUS_MUTE_PARAM;
```

Members

dwPChannel
Performance channel to mute or remap. If the structure is being passed to a get method, this member must be initialized.

dwPChannelMap
Channel to which **dwPChannel** is being mapped. This member is ignored if **fMute** is TRUE.

fMute
TRUE if **dwPChannel** is being muted.

Remarks

If you wanted all the notes on PChannel 3 to play on PChannel 9 instead, you would set **dwPChannel** to 3 and **dwPChannelMap** to 9 before passing the structure to one of the set methods. If you wanted to mute the notes on PChannel 8, you would set **dwPChannel** to 8 and **dwPChannelMap** to 0xFFFFFFFF.

Requirements

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::GetParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicTrack8::SetParamEx**

DMUS_NOTERANGE

Specifies a range of notes that an instrument must respond to. An array of these structures is passed to the **IDirectMusicPerformance8::DownloadInstrument** and **IDirectMusicPort8::DownloadInstrument** methods to specify what notes the instrument should respond to and, therefore, what instrument regions need to be downloaded.

```
typedef struct _DMUS_NOTERANGE {
    DWORD dwLowNote;
    DWORD dwHighNote;
} DMUS_NOTERANGE, *LPDMUS_NOTERANGE;
```

Members

dwLowNote
Low note for this range of MIDI notes to which the instrument must respond.

dwHighNote
High note for this range of MIDI notes to which the instrument must respond.

Requirements

Header: Declared in dmdls.h.

DMUS_OBJECTDESC

Used to describe a DirectMusic object. This structure is passed to the **IDirectMusicLoader8::GetObject** method to identify the object that the loader should retrieve from storage. Information about an object is retrieved in this structure by the **IDirectMusicLoader8::EnumObject** and **IDirectMusicObject8::GetDescriptor** methods.

```
typedef struct _DMUS_OBJECTDESC {
    DWORD         dwSize;
    DWORD         dwValidData;
    GUID          guidObject;
    GUID          guidClass;
    FILETIME      ftDate;
    DMUS_VERSION  vVersion;
    WCHAR         wszName[DMUS_MAX_NAME];
    WCHAR         wszCategory[DMUS_MAX_CATEGORY];
    WCHAR         wszFileName[DMUS_MAX_FILENAME];
    LONGLONG      llMemLength;
    LPBYTE        pbMemData;
    IStream       *pStream
} DMUS_OBJECTDESC, *LPDMUS_OBJECTDESC;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_OBJECTDESC)** before the structure is passed to any method.

dwValidData
Flags describing which members are valid and giving further information about some members. The following values are defined:

DMUS_OBJ_CATEGORY
The **wszCategory** member is valid.

DMUS_OBJ_CLASS

The **guidClass** member is valid.

DMUS_OBJ_DATE
The **ftDate** member is valid.

DMUS_OBJ_FILENAME
The **wszFileName** member is valid. The presence of this flag is assumed if DMUS_OBJ_FULLPATH is set.

DMUS_OBJ_FULLPATH
The **wszFileName** member contains either the full path of a file or a path relative to the application directory. The directory set by **IDirectMusicLoader8::SetSearchDirectory** is not searched. If this flag is not set, **wszFilename** is always assumed to be relative to the application directory, or to the search directory if **SetSearchDirectory** has been called for this object type.

DMUS_OBJ_LOADED
The object is currently loaded in memory.

DMUS_OBJ_MEMORY
The object is in memory, and **llMemLength** and **pbMemData** are valid.

DMUS_OBJ_NAME
The **wszName** member is valid.

DMUS_OBJ_OBJECT
The **guidObject** member is valid.

DMUS_OBJ_STREAM
The **pStream** member contains a pointer to the data stream.

DMUS_OBJ_URL
The **wszFileName** member contains a URL. URLs are not currently supported by the DirectMusic loader.

DMUS_OBJ_VERSION
The **vVersion** member is valid.

guidObject
Unique identifier for this object.

guidClass
Unique identifier for the class of object. All the standard objects have defined identifiers consisting of "CLSID_" plus the name of the object. For example, a segment object is identified as CLSID_DirectMusicSegment. See the defines in the Dmusici.h header file.

ftDate
Date that the object was last edited.

vVersion
DMUS_VERSION structure containing version information.

wszName
Name of the object.

wszCategory
Category for the object.

wszFileName
File path. If DMUS_OBJ_FULLPATH is set, this is the full path; otherwise, it is the file name. If the **IDirectMusicLoader8::SetSearchDirectory** method has been called, this member must contain only a file name.

llMemLength
Size of data in memory.

pbMemData
Data in memory.

pStream
Address of the **IStream** interface of a custom stream that can be used to load the object into memory. In most cases this value should be NULL.See Remarks.

Remarks

At least one of **wszName**, **guidObject**, and **wszFileName** must be filled with valid data to retrieve the object by using the **IDirectMusicLoader8::GetObject** method.

The name and category strings use 16-bit characters in the **WCHAR** format, not 8-bit ANSI characters. Be sure to convert as appropriate. You can use the C library **mbstowcs** function to convert from multibyte to Unicode and the **wcstombs** function to convert from Unicode back to multibyte.

Instead of passing on object descriptor to **IDirectMusicLoader8::GetObject** or **IDirectMusicLoader8::SetObject** with a filename or memory pointer, an application can pass a stream. This is done by setting the DMUS_OBJ_STREAM flag in **dwValidData** and a pointer to the stream in **pStream**. When the application calls **GetObject**, the loader saves the stream's current location, reads the object from the stream, and then restores the saved location. The application can continue reading from the stream without being affected by the call to **GetObject**.

When **SetObject** is called with a stream, the loader makes a copy of the stream which is used if the object is later loaded. Thus an application can release a stream or continue to read from it after passing it to the loader by using **SetObject**.

Requirements

Header: Declared in dmusici.h.

DMUS_PLAY_MARKER_PARAM

Contains information about a play marker.

```
typedef struct _DMUS_PLAY_MARKER_PARAM {
    MUSIC_TIME mtTime;
} DMUS_PLAY_MARKER_PARAM;
```

Members

mtTime
Time of the first legal segment play marker before or at the requested time. The value is an offset from the requested time.

Requirements

Header: Declared in dmusici.h.

See Also

GUID_Play_Marker

DMUS_PORTCAPS

Contains information about a port enumerated by a call to the **IDirectMusic8::EnumPort** method. The structure is also used to return information through the **IDirectMusicPort8::GetCaps** method.

```
typedef struct _DMUS_PORTCAPS {
    DWORD dwSize;
    DWORD dwFlags;
    GUID  guidPort;
    DWORD dwClass;
    DWORD dwType;
    DWORD dwMemorySize;
    DWORD dwMaxChannelGroups;
    DWORD dwMaxVoices;
    DWORD dwMaxAudioChannels;
    DWORD dwEffectFlags;
    WCHAR wszDescription[DMUS_MAX_DESCRIPTION];
} DMUS_PORTCAPS, *LPDMUS_PORTCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_PORTCAPS)** before the structure is passed to any method.

dwFlags
Flags describing various capabilities of the port. This field can contain one or more of the following values:

DMUS_PC_AUDIOPATH
Multiple outputs can be connected to DirectSound for audiopaths.

DMUS_PC_DIRECTSOUND
The port supports streaming wave data to DirectSound.

DMUS_PC_DLS
The port supports DLS Level 1 sample collections.

DMUS_PC_DLS2
The port supports DLS Level 2 sample collections.

DMUS_PC_EXTERNAL
The port connects to devices outside the host—for example, devices connected over an external MIDI port such as the MPU-401.

DMUS_PC_GMINHARDWARE
The synthesizer has its own GM instrument set, so GM instruments do not need to be downloaded.

DMUS_PC_GSINHARDWARE
This port contains the Roland GS sound set in hardware.

DMUS_PC_MEMORYSIZEFIXED
Memory available for DLS instruments cannot be adjusted.

DMUS_PC_SHAREABLE
More than one port can be created that uses the same range of channel groups on the device. Unless this bit is set, the port can be opened only in exclusive mode. In exclusive mode, an attempt to create a port fails unless free channel groups are available to assign to the create request.

DMUS_PC_SOFTWARESYNTH
The port is a software synthesizer.

DMUS_PC_WAVE
Streaming and one-shot waves are supported.

DMUS_PC_XGINHARDWARE
The port contains the Yamaha XG extensions in hardware.

guidPort
Identifier of the port. This value can be passed to the **IDirectMusic8::CreatePort** method to get an **IDirectMusicPort8** interface for the port.

dwClass
Class of this port. The following classes are defined:

DMUS_PC_INPUTCLASS
Input port.

DMUS_PC_OUTPUTCLASS
Output port.

dwType
Type of this port. The following types are defined:

DMUS_PORT_WINMM_DRIVER
Windows multimedia driver.

DMUS_PORT_USER_MODE_SYNTH
User-mode synthesizer.

DMUS_PORT_KERNEL_MODE
WDM driver.

dwMemorySize
Amount of memory available to store DLS instruments. If the port is using system memory and the amount is therefore limited only by the available system memory, this member contains DMUS_PC_SYSTEMMEMORY.

dwMaxChannelGroups
Maximum number of channel groups supported by this port. A channel group is a set of 16 MIDI channels.

dwMaxVoices
Maximum number of voices that can be allocated when this port is opened. The value can be –1 if the driver does not support returning this parameter.

dwMaxAudioChannels
Maximum number of audio channels that can be rendered by the port. The value can be –1 if the driver does not support returning this parameter.

dwEffectFlags
Flags indicating what audio effects are available on the port.

The following flags are defined:

DMUS_EFFECT_NONE
No effects are supported.

DMUS_EFFECT_REVERB
The port supports reverb.

DMUS_EFFECT_CHORUS
The port supports chorus.

wszDescription
Description of the port. This can be a system-generated name, such as L"MPU-401 Output Port [330]", or a user-specified friendly name, such as L"Port w/ External SC-55".

Requirements

Header: Declared in dmusicc.h.

DMUS_PORTPARAMS8

Contains parameters for the opening of a DirectMusic port. These parameters are passed in when the **IDirectMusic8::CreatePort** method is called.

The define **DMUS_PORTPARAMS** resolves to **DMUS_PORTPARAMS8**. This structure supersedes the earlier version of **DMUS_PORTPARAMS**, which is now declared as **DMUS_PORTPARAMS7**.

```
typedef struct _DMUS_PORTPARAMS8 {
    DWORD dwSize;
    DWORD dwValidParams;
    DWORD dwVoices;
    DWORD dwChannelGroups;
```

```
    DWORD dwAudioChannels;
    DWORD dwSampleRate;
    DWORD dwEffectFlags;
    DWORD fShare;
    DWORD dwFeatures;
} DMUS_PORTPARAMS8;

typedef DMUS_PORTPARAMS8 DMUS_PORTPARAMS;
typedef DMUS_PORTPARAMS *LPDMUS_PORTPARAMS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_PORTPARAMS8)** before the structure is passed to a method.

dwValidParams
Specifies which members in this structure are valid. Setting the flag for a particular port parameter means that you want to have this parameter set on the method call or want to override the default value when the port is created. The following flags have been defined:

DMUS_PORTPARAMS_VOICES
DMUS_PORTPARAMS_CHANNELGROUPS
DMUS_PORTPARAMS_AUDIOCHANNELS
DMUS_PORTPARAMS_SAMPLERATE
DMUS_PORTPARAMS_EFFECTS
DMUS_PORTPARAMS_SHARE
DMUS_PORTPARAMS_FEATURES

dwVoices
Number of voices required on this port. This is not an absolute maximum; the port can create additional temporary voices to enable smooth transitions when lower-priority voices have to be dropped.

dwChannelGroups
Number of channel groups to be allocated on this port. Must be less than or equal to the number of channel groups specified in the **DMUS_PORTCAPS** structure returned by the **IDirectMusic8::EnumPort** and **IDirectMusicPort8::GetCaps** methods.

dwAudioChannels
Desired number of output channels.

dwSampleRate
Desired sample rate, in hertz.

dwEffectFlags
Flags indicating which special effects are desired. The following flags are defined:

DMUS_EFFECT_NONE
DMUS_EFFECT_REVERB
DMUS_EFFECT_CHORUS

fShare
If TRUE, all ports use the channel groups assigned to this port. If FALSE, the port is opened in exclusive mode, and the use of the same channel groups by other ports is forbidden.

dwFeatures
Miscellaneous capabilities of the port. The following values are defined.

DMUS_PORT_FEATURE_AUDIOPATH
Supports an audiopath connection to DirectSound buffers.

DMUS_PORT_FEATURE_STREAMING
Supports streaming waves through the synthesizer.

Remarks

The **DMUS_PORTPARAMS** structure from versions prior to DirectX 8.0 is maintained in Dmusicc.h as **DMUS_PORTPARAMS7**.

Requirements

Header: Declared in dmusicc.h.

See Also

DMUS_PORTCAPS

DMUS_RHYTHM_PARAM

Used as the *pParam* parameter in calls to the various get-parameter methods when the track is a chord track and *rguidType* is GUID_RhythmParam.

```
typedef struct {
    DMUS_TIMESIGNATURE  TimeSig;
    DWORD               dwRhythmPattern;
} DMUS_RHYTHM_PARAM;
```

Members

TimeSig
DMUS_TIMESIGNATURE structure containing the time signature of the rhythm parameter. This structure must be initialized before the **DMUS_RHYTHM_PARAM** structure is passed to the get method.

dwRhythmPattern

Rhythm pattern for a sequence of chords. Each bit represents a beat in one or more measures, with 1 signifying a chord on the beat and 0 signifying no chord.

Requirements

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::GetParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicTrack8::SetParamEx**

DMUS_SCRIPT_ERRORINFO

Contains information about a script error.

```
typedef struct _DMUS_SCRIPT_ERRORINFO {
    DWORD   dwSize;
    HRESULT hr;
    ULONG ulLineNumber;
    LONG ichCharPosition;
    WCHAR  wszSourceFile[DMUS_MAX_FILENAME];
    WCHAR  wszSourceComponent[DMUS_MAX_FILENAME];
    WCHAR  wszDescription[DMUS_MAX_FILENAME];
    WCHAR  wszSourceLineText[DMUS_MAX_FILENAME];
} DMUS_SCRIPT_ERRORINFO;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is passed to any of the **IDirectMusicScript8** methods.

hr
Result code obtained from DirectMusic or the script engine.

ulLineNumber
Line number in the script where the error occurred.

ichCharPosition
Position in the line where a syntax error was found, if **wszSourceLineText** contains a string.

wszSourceFile
File name of the script.

wszSourceComponent
Name of the component that generated the error. For example, this could be DirectMusic or the script parsing engine.

wszDescription
Description of the error.

wszSourceLineText
Text of the script line where a syntax error occurred. If the error is not in the syntax, this is an empty string.

Requirements

Header: Declared in dmusici.h.

DMUS_SUBCHORD

Used in the **SubChordList** member of a **DMUS_CHORD_PARAM** structure.

```
typedef struct {
    DWORD dwChordPattern;
    DWORD dwScalePattern;
    DWORD dwInversionPoints;
    DWORD dwLevels;
    BYTE  bChordRoot;
    BYTE  bScaleRoot;
} DMUS_SUBCHORD;
```

Members

dwChordPattern
Notes in the subchord. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the chord.

dwScalePattern
Notes in the scale. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the scale.

dwInversionPoints
Points in the scale at which inversions can occur. Bits that are off signify that the notes in the interval cannot be inverted. Thus, the pattern 100001111111 indicates that inversions are allowed anywhere except between the fifth and seventh degrees of a major scale.

dwLevels
Bit field showing which levels are supported by this subchord. Each part in a style is assigned a level, and this chord is used only for parts whose levels are contained in this member.

bChordRoot
Root of the subchord, in which 0 is the lowest C in the range and 23 is the top B.

bScaleRoot
Root of the scale, in which 0 is the lowest C in the range and 23 is the top B.

Requirements

Header: Declared in dmusici.h.

DMUS_SYNTHSTATS8

Used by the **IDirectMusicPort8::GetRunningStats** method to return the current running status of a synthesizer.

```
typedef struct DMUS_SYNTHSTATS {
    DWORD dwSize;
    DWORD dwValidStats;
    DWORD dwVoices;
    DWORD dwTotalCPU;
    DWORD dwCPUPerVoice;
    DWORD dwLostNotes;
    DWORD dwFreeMemory;
    long  lPeakVolume;
        DWORD   dwSynthMemUse;
} DMUS_SYNTHSTATS8;

typedef struct _DMUS_SYNTHSTATS8 *LPDMUS_SYNTHSTATS8;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_SYNTHSTATS)** before the structure is passed to a method.

dwValidStats
Flags that specify which fields in this structure have been filled in by the synthesizer. The following flags have been defined:

DMUS_SYNTHSTATS_VOICES
DMUS_SYNTHSTATS_TOTAL_CPU
DMUS_SYNTHSTATS_CPU_PER_VOICE
DMUS_SYNTHSTATS_FREE_MEMORY
DMUS_SYNTHSTATS_LOST_NOTES
DMUS_SYNTHSTATS_PEAK_VOLUME

dwVoices
Average number of voices playing.

dwTotalCPU
Total percentage of the CPU being consumed, multiplied by 100.

dwCPUPerVoice
Percentage of the CPU being consumed per voice, multiplied by 100.

dwLostNotes

Number of notes lost. Notes can be dropped because of voice-stealing or because too much of the CPU is being consumed.

dwFreeMemory
Amount of memory currently available to store DLS instruments. If the synthesizer is using system memory and the amount is therefore limited only by the available system memory, this value is set to DMUS_SYNTHSTATS_SYSTEMMEMORY.

lPeakVolume
Peak volume, measured in hundredths of decibels.

dwSynthMemUse
Memory used by synthesizer wave data.

Remarks

All the running status parameters, with the exception of **dwFreeMemory**, are refreshed every second. For example, **dwLostNotes** provides the total number of notes lost over a one-second period.

Requirements

Header: Declared in dmusici.h.

DMUS_TEMPO_PARAM

Used as the *pParam* parameter in calls to the various get-parameter and set-parameter methods when the track is a tempo track and *rguidType* is GUID_TempoParam.

```
typedef struct _DMUS_TEMPO_PARAM {
    MUSIC_TIME  mtTime;
    double      dblTempo;
} DMUS_TEMPO_PARAM;
```

Members

mtTime
Time for which the tempo was retrieved. (This member is not used in set methods, which use their *mtTime* parameter instead.)

dblTempo
The tempo, in the range from DMUS_TEMPO_MIN through DMUS_TEMPO_MAX.

Requirements

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::GetParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicTrack8::SetParamEx**

DMUS_TIMESIGNATURE

Used by the **IDirectMusicStyle8::GetTimeSignature** method to retrieve information about a style's time signature. It is also used in the **DMUS_RHYTHM_PARAM** structure and in the various get-parameter methods when the *rguidType* parameter is GUID_TimeSignature and the track is a time signature or style track.

```
typedef struct _DMUS_TIMESIGNATURE {
    MUSIC_TIME mtTime;
    BYTE  bBeatsPerMeasure;
    BYTE  bBeat;
    WORD  wGridsPerBeat;
} DMUS_TIMESIGNATURE;
```

Members

mtTime
Music time at which this time signature occurs.

bBeatsPerMeasure
Top of time signature.

bBeat
Bottom of time signature.

wGridsPerBeat
Grids (subdivisions) per beat. This value determines the timing resolution for certain music events—for example, segments cued with the DMUS_SEGF_GRID flag (see **DMUS_SEGF_FLAGS**).

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicPerformance8::SetParam**, **IDirectMusicSegment8::GetParam**, **IDirectMusicSegment8::SetParam**, **IDirectMusicTrack8::GetParamEx**, **IDirectMusicTrack8::SetParamEx**, **DMUS_TIMESIG_PMSG**

DMUS_VALID_START_PARAM

Used as the *pParam* parameter in calls to various get-parameter methods when *rguidType* is GUID_Valid_Start_Time.

```
typedef struct _DMUS_VALID_START_PARAM {
    MUSIC_TIME mtTime;
} DMUS_VALID_START_PARAM;
```

Members

mtTime
Next valid point at which the segment can start.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::GetParam, **IDirectMusicSegment8::GetParam**, **IDirectMusicTrack8::GetParamEx**

DMUS_VARIATIONS_PARAM

Contains information about variations associated with channels. It is used when retrieving the GUID_Variations parameter.

```
typedef struct _DMUS_VARIATIONS_PARAM {
    DWORD   dwPChannelsUsed;
    DWORD*  padwPChannels;
    DWORD*   padwVariations;
} DMUS_VARIATIONS_PARAM;
```

Members

dwPChannelsUsed
DWORD that receives the number of performance channels in use.

padwPChannels
Address of an array of **DWORD** values that receives the performance channels in use.

padwVariations
Array of variations in effect for each channel, where each bit set represents a variation.

Requirements

Header: Declared in dmusici.h.

DMUS_VERSION

Contains version information for an object described in the **DMUS_OBJECTDESC** structure.

```
typedef struct _DMUS_VERSION {
    DWORD  dwVersionMS;
    DWORD  dwVersionLS;
} DMUS_VERSION, FAR *LPDMUS_VERSION;
```

Members

dwVersionMS
Most significant **DWORD** of the version number.

dwVersionLS
Least significant **DWORD** of the version number.

Requirements

Header: Declared in dmusici.h.

DMUS_WAVES_REVERB_PARAMS

Contains information about reverberation effects in the Microsoft software synthesizer provided with DirectX 7.0.

```
typedef struct _DMUS_WAVES_REVERB_PARAMS {
    float  fInGain;
    float  fReverbMix;
    float  fReverbTime;
    float  fHighFreqRTRatio;
} DMUS_WAVES_REVERB_PARAMS;
```

Members

fInGain
Input gain, in decibels (to avoid output overflows). The default value is 0.

fReverbMix
Reverb mix, in decibels. A value of 0 means 100 percent wet reverb (no direct signal). Negative values gives less wet signal. The coefficients are calculated so

that the overall output level stays approximately constant, regardless of the amount of reverb mix. The default value is –10.0.

fReverbTime
Reverb decay time, in milliseconds. The default value is 1000.

fHighFreqRTRatio
Ratio of the high frequencies to the global reverb time. Unless very bright reverbs are wanted, this should be set to a value less than 1. For example, if **fReverbTime** is 1000 ms and **dHighFreqRTRatio** is 0.1, the decay time for high frequencies is 100 ms. The default value is 0.001.

Remarks

The TrueVerb reverberation technology from Waves is licensed to Microsoft as the SimpleVerb implementation for use in the Microsoft software synthesizer.

In DirectX 8.0 and later, music reverberation is handled by a DMO. For more information, see Waves Reverberation.

Requirements

Header: Declared in dmusicc.h.

KSPROPERTY

Used by the **IKsControl::KsProperty** method to identify a property and operation.

KSPROPERTY is defined as a **KSIDENTIFIER** structure, which is declared as follows:

```
typedef struct {
    union {
        struct {
            GUID   Set;
            ULONG  Id;
            ULONG  Flags;
        };
        LONGLONG    Alignment;
    };
} KSIDENTIFIER, *PKSIDENTIFIER;
```

Members

Set
Identifier of the property set. The following property-set GUIDs are predefined by DirectMusic:

GUID_DMUS_PROP_DLS1

Item 0 is a Boolean indicating whether or not this port supports downloading DLS level 1 samples.

GUID_DMUS_PROP_DLS2
Item 0 is a Boolean indicating whether or not this port supports downloading DLS level 2 samples.

GUID_DMUS_PROP_Effects
Item 0 contains DMUS_EFFECT_NONE or one or more effects flags (see the **dwEffectFlags** member of **DMUS_PORTCAPS**). This property is used to set or retrieve the current state of the effects.

GUID_DMUS_PROP_GM_Hardware
Item 0 is a Boolean indicating whether or not this port supports GM in hardware.

GUID_DMUS_PROP_GS_Capable
Item 0 is a Boolean indicating whether or not this port supports the minimum requirements for Roland GS extensions.

GUID_DMUS_PROP_GS_Hardware
Item 0 is a Boolean indicating whether or not this port supports Roland GS extensions in hardware.

GUID_DMUS_PROP_INSTRUMENT2
Item 0 is a Boolean indicating whether or not this port supports downloading samples using the **DMUS_ARTICULATION2** structure.

GUID_DMUS_PROP_LegacyCaps
Item 0 is the **MIDIINCAPS** or **MIDIOUTCAPS** structure that describes the underlying Windows multimedia device implementing this port. A **MIDIINCAPS** structure is returned if **dwClass** is DMUS_PC_INPUTCLASS in this port's capabilities structure. Otherwise, a **MIDIOUTCAPS** structure is returned.

GUID_DMUS_PROP_MemorySize
Item 0 is the number of bytes of sample RAM free on this device.

GUID_DMUS_PROP_SampleMemorySize
Item 0 is the number of bytes of sample RAM, both free and used, available on this device.

GUID_DMUS_PROP_SamplePlaybackRate
Item 0 is the synthesizer's sample rate. The DLS level 2 file format supports conditional chunks to determine whether a region or articulation should be downloaded. This allows authors to create optional waves intended for different sample rates.

GUID_DMUS_PROP_SynthSink_DSOUND
Item 0 is a Boolean indicating whether or not this port supports DirectSound.

GUID_DMUS_PROP_SynthSink_WAVE
Item 0 is a Boolean indicating whether or not this port supports wave output using the **waveOut** functions.

GUID_DMUS_PROP_Volume

Item 1 (DMUS_ITEM_Volume) is a **LONG** in the range from DMUS_VOLUME_MAX through DMUS_VOLUME_MIN. This is the signed value, in hundredths of a decibel, which is added to the gain of all voices after all DLS articulation has been performed. By default, when a port is added to the performance, this property is set to the master volume. For master volume, see Setting and Retrieving Global Parameters.

GUID_DMUS_PROP_WavesReverb
Item 0 is a **DMUS_WAVES_REVERB_PARAMS** structure containing reverb parameters.

GUID_DMUS_PROP_WriteLatency
Item 0 is the write latency of the user-mode synthesizer (the **dwType** member of the **DMUS_PORTCAPS** structure is DMUS_PORT_USER_MODE_SYNTH) that streams its output to DirectSound. The write latency is the delay between when the synthesizer creates a buffer of sound and when it is heard. By adjusting this value, an application can fine-tune the synthesizer for minimum latency without sound breakup. On some computers, in particular, ones without hardware support for DirectSound, the initial latency is much larger than on others, so the value should always be read first, and then adjusted with a relative value. The write latency can have different values for each port instance. The property must be set each time the port is activated. It is not recommended that applications set the write latency of a synthesizer in an audiopath.

GUID_DMUS_PROP_WritePeriod
Item 0 is the write period, in milliseconds, of the user-mode synthesizer (the **dwType** member of the **DMUS_PORTCAPS** structure is DMUS_PORT_USER_MODE_SYNTH) that streams its output to DirectSound. The write period controls how frequently the synthesizer sink allows the synthesizer to mix. By reducing this value, the application can reduce the overall latency of the synthesizer. However, values under 10 milliseconds increase the CPU load. The write period has the same value for all port instances that use the standard DirectSound sink. The property must be set each time the port is activated. This property cannot be set on ports in audiopaths.

GUID_DMUS_PROP_XG_Capable
Item 0 is a Boolean indicating whether or not this port supports the minimum requirements for Yamaha XG extensions.

GUID_DMUS_PROP_XG_Hardware
Item 0 is a Boolean indicating whether or not this port supports Yamaha XG extensions in hardware.

Id

Item within the property set.

Flags

One of the following flags to specify the operation:

KSPROPERTY_TYPE_GET
To retrieve the given property item's value.

KSPROPERTY_TYPE_SET
To set the given property item's value.

KSPROPERTY_TYPE_BASICSUPPORT
To determine the type of support available for the property set. The data returned by **IKsControl::KsProperty** in **pvPropertyData* is a **DWORD** containing one or both of KSPROPERTY_TYPE_GET and KSPROPERTY_TYPE_SET, indicating which operations are possible.

Alignment
Not used in DirectMusic.

Requirements

Header: Declared in dmksctrl.h.

See Also

Property Sets for DirectMusic Ports

DLS Structures

This section contains reference information for structures used with Downloadable Sounds. Most applications do not need to use these structures, because DirectMusic handles the details of loading DLS collections and downloading instruments to the synthesizer. They are of interest chiefly for applications that edit DLS.

For an overview of using DLS data, see Low-Level DLS.

For more information on DLS data formats, see the specification from the MIDI Manufacturers Association.

The following structures are included in this section:

- **DMUS_ARTICPARAMS**
- **DMUS_ARTICULATION**
- **DMUS_ARTICULATION2**
- **DMUS_COPYRIGHT**
- **DMUS_DOWNLOADINFO**
- **DMUS_EXTENSIONCHUNK**
- **DMUS_INSTRUMENT**
- **DMUS_LFOPARAMS**
- **DMUS_MSCPARAMS**
- **DMUS_OFFSETTABLE**
- **DMUS_PEGPARAMS**
- **DMUS_REGION**

- **DMUS_VEGPARAMS**
- **DMUS_WAVE**
- **DMUS_WAVEARTDL**
- **DMUS_WAVEDATA**
- **DMUS_WAVEDL**

DMUS_ARTICPARAMS

Describes parameters for a DLS level 1 articulation chunk. All parameters for articulation are stored in one chunk, which comprises a series of structures defining each functional area of the articulation. If an instrument or region uses articulation, it references this chunk by index from the **DMUS_ARTICULATION** chunk.

```
typedef struct {
    DMUS_LFOPARAMS LFO;
    DMUS_VEGPARAMS VolEG;
    DMUS_PEGPARAMS PitchEG;
    DMUS_MSCPARAMS Misc;
} DMUS_ARTICPARAMS;
```

Members

LFO
DMUS_LFOPARAMS structure containing parameters for a low-frequency oscillator.

VolEG
DMUS_VEGPARAMS structure containing parameters for a volume-envelope generator.

PitchEG
DMUS_PEGPARAMS structure containing parameters for a pitch-envelope generator.

Misc
DMUS_MSCPARAMS structure containing the initial pan position.

Remarks

DLS level 2 articulation is handled differently and does not use this structure. See **DMUS_ARTICULATION2**.

Requirements

Header: Declared in dmdls.h.

DMUS_ARTICULATION

Describes a DLS instrument articulation chunk. It is used when the format identifier in the **dwDLType** member of the **DMUS_DOWNLOADINFO** structure is DMUS_DOWNLOADINFO_INSTRUMENT. This chunk connects all available DLS articulation data in one list. For example, it might have a DLS Level 1 chunk and a manufacturer's proprietary articulation chunk. The DLS chunk is referenced by **ulArt1Idx**, and all additional articulation chunks are referenced by the list that starts with **ulFirstExtCkIdx**.

```
typedef struct {
    ULONG ulArt1Idx;
    ULONG ulFirstExtCkIdx;
} DMUS_ARTICULATION;
```

Members

ulArt1Idx
Index, in the **DMUS_OFFSETTABLE** structure, of the DLS articulation chunk. If 0, there is no DLS articulation.

ulFirstExtCkIdx
Index of the first third-party extension chunk. If 0, there are no third-party extension chunks associated with the articulation.

Remarks

The articulation chunk consists of a **DMUS_ARTICPARAMS** structure.

Requirements

Header: Declared in dmdls.h.

See Also

DMUS_ARTICULATION2

DMUS_ARTICULATION2

Describes a DLS instrument articulation chunk. This structure is used when the format identifier in the **dwDLType** member of the **DMUS_DOWNLOADINFO** structure is DMUS_DOWNLOADINFO_INSTRUMENT2. The DLS level 1 chunk is referenced by **ulArt1Idx**, and all additional articulation chunks are referenced by the list that starts with **ulFirstExtCkIdx**. DLS level 2 articulation chunks also use **ulNextArtIdx**.

```
typedef struct {
    ULONG ulArt1Idx;
    ULONG ulFirstExtCkIdx;
    ULONG ulNextArtIdx;
} DMUS_ARTICULATION;
```

Members

ulArt1Idx
Index, in the **DMUS_OFFSETTABLE** structure, of the DLS articulation chunk. If 0, there is no DLS level 1 or 2 articulation.

ulFirstExtCkIdx
Index of the first third-party extension chunk. If 0, there are no third-party extension chunks associated with the articulation. DLS level 2 chunks can also be placed here.

ulNextArtIdx
Index of additional articulation chunks to better support DLS level 2 articulations.

Remarks

The articulation chunk consists of a **CONNECTIONLIST** structure followed by an array of **CONNECTION** structures. These structures are declared in Dls1.h. For more information, see the Downloadable Sounds Level 2 specification, published by the MIDI Manufacturers Association.

Requirements

Header: Declared in dmdls.h.

See Also

DMUS_ARTICULATION

DMUS_COPYRIGHT

Describes an optional copyright chunk in DLS data.

```
typedef struct {
    ULONG cbSize;
    BYTE  byCopyright[ ];
} DMUS_COPYRIGHT;
```

Members

cbSize
Size of data.

byCopyright[ ]
Copyright data.

Requirements

Header: Declared in dmdls.h.

DMUS_DOWNLOADINFO

Used as a header for DLS data to be downloaded to a port. It defines the size and functionality of the download and is always followed by a **DMUS_OFFSETTABLE** chunk.

```
typedef struct _DMUS_DOWNLOADINFO {
    DWORD dwDLType;
    DWORD dwDLId;
    DWORD dwNumOffsetTableEntries;
    DWORD cbSize;
} DMUS_DOWNLOADINFO;
```

Members

dwDLType
Type of data being downloaded. The following types are defined:

DMUS_DOWNLOADINFO_INSTRUMENT
Instrument definition, starting with the **DMUS_INSTRUMENT** structure.

DMUS_DOWNLOADINFO_INSTRUMENT2
Instrument definition supporting DLS level 2 articulation, starting with the **DMUS_INSTRUMENT** structure.

DMUS_DOWNLOADINFO_WAVE
PCM wave data, starting with the **DMUS_WAVE** structure.

dwDLId
Unique 32-bit identifier for the object. See Remarks.

dwNumOffsetTableEntries
Number of entries in the **DMUS_OFFSETTABLE** structure that follows.

cbSize
Total size of **DMUS_DOWNLOADINFO**, **DMUS_OFFSETTABLE**, and the actual data chunk.

Remarks

The identifier in **dwDLId** is used to connect objects and is obtained by using the **IDirectMusicPortDownload8::GetDLId** method. Primarily it connects the regions in an instrument to wave chunks. For example, if a wave download is given a **dwDLId** of 3, an instrument chunk downloads with the value 3 placed in the **WaveLink.ulTableIndex** member of one of its **DMUS_REGION** structures. This indicates that the region is connected to the wave chunk.

Requirements

Header: Declared in dmdls.h.

See Also

Low-Level DLS

DMUS_EXTENSIONCHUNK

Describes a DLS extension chunk. All extensions to the DLS file format that are unknown to DirectMusic are downloaded in this variable-size chunk.

```
typedef struct {
    ULONG  cbSize;
    ULONG  ulNextExtCkIdx;
    FOURCC ExtCkID;
    BYTE   byExtCk[ ];
} DMUS_EXTENSIONCHUNK;
```

Members

cbSize
Size of chunk.

ulNextExtCkIdx
Index, in the **DMUS_OFFSETTABLE** structure, of the next extension chunk. If 0, there are no more third-party extension chunks.

ExtCkID
Chunk identifier.

byExtCk[ ]
Data.

Requirements

Header: Declared in dmdls.h.

DMUS_INSTRUMENT

Contains an instrument definition in a DLS download chunk.

```
typedef struct {
    ULONG ulPatch;
    ULONG ulFirstRegionIdx;
    ULONG ulGlobalArtIdx;
    ULONG ulFirstExtCkIdx;
    ULONG ulCopyrightIdx;
    ULONG ulFlags;
} DMUS_INSTRUMENT;
```

Members

ulPatch
Patch number of instrument.

ulFirstRegionIdx
Index of first region chunk (see **DMUS_REGION**) within the instrument. There should always be a region, but for compatibility with future synthesizer architectures, it is acceptable to have 0 in this member.

ulGlobalArtIdx
Index, in the **DMUS_OFFSETTABLE** structure, of the global articulation chunk (see **DMUS_ARTICULATION** and **DMUS_ARTICULATION2**) for the instrument. If 0, the instrument does not have global articulation.

ulFirstExtCkIdx
Index, in the **DMUS_OFFSETTABLE** structure, of the first extension chunk (see **DMUS_EXTENSIONCHUNK**) within the instrument. This is used to add new chunks that DirectMusic is unaware of. If 0, no third-party extension chunks are associated with the instrument.

ulCopyrightIdx
Index, in the **DMUS_OFFSETTABLE** structure, of an optional copyright chunk (see **DMUS_COPYRIGHT**). If 0, no copyright information is associated with the instrument.

ulFlags
Additional flags for the instrument. The following flag is defined:

DMUS_INSTRUMENT_GM_INSTRUMENT
The instrument is a standard General MIDI instrument. In the case of patch overlap, GM instruments always have lower priority than other DLS instruments. For example, if a GM instrument is downloaded with patch 0 and a non-GM instrument is also downloaded at patch 0, the non-GM instrument is always selected for playback.

Requirements

Header: Declared in dmdls.h.

DMUS_LFOPARAMS

Defines the low-frequency oscillator for a DLS level 1 articulation chunk. It is used in the **DMUS_ARTICPARAMS** structure.

```
typedef struct {
    PCENT pcFrequency;
    TCENT tcDelay;
    GCENT gcVolumeScale;
    PCENT pcPitchScale;
    GCENT gcMWToVolume;
    PCENT pcMWToPitch;
} DMUS_LFOPARAMS;
```

Members

pcFrequency
Frequency, in pitch units. See Remarks.

tcDelay
Initial delay, in time cents. See Remarks.

gcVolumeScale
Scaling of output to control tremolo, in attenuation units. See Remarks.

pcPitchScale
Scaling of LFO output to control vibrato, in pitch units. See Remarks.

gcMWToVolume
Modulation wheel range to control tremolo, in attenuation units. See Remarks.

pcMWToPitch
Modulation wheel range to control tremolo, in attenuation units. See Remarks.

Remarks

The DLS Level 1 specification defines time cents, pitch cents, and attenuation as 32-bit logarithmic values. See the specification from the MIDI Manufacturers Association for details.

Requirements

Header: Declared in dmdls.h.

See Also

DMUS_ARTICPARAMS

DMUS_MSCPARAMS

Defines the pan for a DLS level 1 articulation chunk. This structure is used in the **DMUS_ARTICPARAMS** structure.

```
typedef struct {
    PERCENT ptDefaultPan;
} DMUS_MSCPARAMS;
```

Members

ptDefaultPan
Default pan, ranging from –50 through 50 percent, in units of 0.1 percent shifted left by 16.

Remarks

PERCENT is defined as **long**. For more information about pan values, see the DLS specification from the MIDI Manufacturers Association.

Requirements

Header: Declared in dmdls.h.

See Also

DMUS_ARTICPARAMS

DMUS_OFFSETTABLE

Used in the header of DLS instrument data being downloaded to a port.

```
typedef struct _DMUS_OFFSETTABLE {
    ULONG ulOffsetTable[DMUS_DEFAULT_SIZE_OFFSETTABLE];
} DMUS_OFFSETTABLE;
```

Members

ulOffsetTable
Array of byte offsets into the data.

Requirements

Header: Declared in dmdls.h.

See Also

Low-Level DLS

DMUS_PEGPARAMS

Defines the pitch envelope for a DLS level 1 articulation chunk. It is used in the **DMUS_ARTICPARAMS** structure.

```
typedef struct {
    TCENT   tcAttack;
    TCENT   tcDecay;
    PERCENT ptSustain;
    TCENT   tcRelease;
    TCENT   tcVel2Attack;
    TCENT   tcKey2Decay;
    PCENT   pcRange;
} DMUS_PEGPARAMS;
```

Members

tcAttack
Attack time, in time cents. See Remarks.

tcDecay
Decay time, in time cents. See Remarks.

ptSustain
Sustain, in hundredths of a percent shifted left by 16.

tcRelease
Release time, in time cents. See Remarks.

tcVel2Attack
Velocity to attack, in time cents. See Remarks.

tcKey2Decay
Key to decay, in time cents. See Remarks.

pcRange
Envelope range, in pitch units. See Remarks.

Remarks

The DLS Level 1 specification defines time cents and pitch cents as 32-bit logarithmic values. See the specification from the MIDI Manufacturers Association for details about the values in this structure.

Requirements

Header: Declared in dmdls.h.

See Also

DMUS_ARTICPARAMS

DMUS_REGION

Defines a region for a DLS download. One or more regions can be embedded in an instrument buffer and referenced by the instrument header chunk, **DMUS_INSTRUMENT**.

```
typedef struct {
    RGNRANGE RangeKey;
    RGNRANGE RangeVelocity;
    USHORT   fusOptions;
    USHORT   usKeyGroup;
    ULONG    ulRegionArtIdx;
    ULONG    ulNextRegionIdx;
    ULONG    ulFirstExtCkIdx;
    WAVELINK WaveLink;
    WSMPL    WSMP;
    WLOOP    WLOOP[1];
} DMUS_REGION;
```

Members

RangeKey
Key range for this region.

RangeVelocity
Velocity range for this region.

fusOptions
Options for the synthesis of this region. The following flag is defined:

F_RGN_OPTION_SELFNONEXCLUSIVE
If a second note-on for the same note is received by the synthesis engine, the second note is played, as well as the first. This option is off by default so that the synthesis engine forces a note-off of the first note.

usKeyGroup
Key group for a drum instrument. Key group values allow multiple regions within a drum instrument to belong to the same group. If a synthesis engine is instructed to play a note with a key group setting and any other notes are currently playing with this same key group, the synthesis engine turns off all notes with the same key group value as soon as possible. Currently, key groups from 1 through 15 are legal, and 0 indicates no key group.

ulRegionArtIdx
Index, in the **DMUS_OFFSETTABLE** structure, of the global articulation chunk for the region. If 0, the region does not have an articulation and relies on the instrument's global articulation.

ulNextRegionIdx
Index, in the **DMUS_OFFSETTABLE** structure, of the next region in the region list. If 0, there are no more regions.

ulFirstExtCkIdx
Index, in the **DMUS_OFFSETTABLE** structure, of the third-party extension chunk list. If 0, no extension chunks are associated with the region.

WaveLink
Standard DLS structure (declared in the Dls1.h header file) for managing a link from the region to a wave. The **ulTableIndex** member of the **WAVELINK** structure contains the download identifier of the associated wave buffer. (For more information, see **DMUS_DOWNLOADINFO** and Low-Level DLS.)

WSMP
Standard DLS structure (declared in Dls1.h) for managing the playback of the wave. If the **cSampleLoops** member is 1, the following **WLOOP** structure carries the loop start and end points.

WLOOP[ ]
Standard DLS structure (declared in Dls1.h) for describing a loop.

Requirements

Header: Declared in dmdls.h.

DMUS_VEGPARAMS

Defines a volume envelope for a DLS level 1 articulation chunk.

```
typedef struct {
    TCENT   tcAttack;
    TCENT   tcDecay;
    PERCENT ptSustain;
    TCENT   tcRelease;
    TCENT   tcVel2Attack;
    TCENT   tcKey2Decay;
} DMUS_VEGPARAMS;
```

Members

tcAttack
Attack time, in time cents. See Remarks.

tcDecay
Decay time, in time cents. See Remarks.

ptSustain
Sustain, in hundredths of a percent and shifted left by 16.

tcRelease
Release time, in time cents. See Remarks.

tcVel2Attack
Velocity to attack, in time cents. See Remarks.

tcKey2Decay
Key to decay, in time cents. See Remarks.

Remarks

The DLS Level 1 specification defines time cents as a 32-bit logarithmic value. See the specification from the MIDI Manufacturers Association for details about the values in this structure.

Requirements

Header: Declared in dmdls.h.

See Also

DMUS_ARTICPARAMS

DMUS_WAVE

Defines a wave chunk for a DLS download.

```
typedef struct {
    ULONG  ulFirstExtCkIdx;
    ULONG  ulCopyrightIdx;
    ULONG  ulWaveDataIdx;
    WAVEFORMATEX  WaveformatEx;
} DMUS_WAVE;
```

Members

ulFirstExtCkIdx
Index, in the **DMUS_OFFSETTABLE** structure, of third-party extension chunks. If 0, no extension chunks are associated with the wave.

ulCopyrightIdx
Index, in the **DMUS_OFFSETTABLE** structure, of copyright chunks. If 0, no copyright information is associated with the wave.

ulWaveDataIdx
Index, in the **DMUS_OFFSETTABLE** structure, of wave data. See **DMUS_WAVEDATA**.

WaveformatEx
WAVEFORMATEX structure that specifies the wave format of the chunk.

Requirements

Header: Declared in dmdls.h.

DMUS_WAVEARTDL

Contains information for downloading wave articulation.

```
typedef struct _DMUS_WAVEARTDL {
    ULONG   ulDownloadIdIdx;
    ULONG   ulBus;
    ULONG   ulBuffers;
    ULONG   ulMasterDLId;
    USHORT  usOptions
}   DMUS_WAVEARTDL, *LPDMUS_WAVEARTDL;
```

Members

ulDownloadIdIdx
Download identifiers of each buffer.

ulBus
Playback bus.

ulBuffers
Buffers.

ulMasterDLId
Download identifier of master voice of subordinate group.

usOptions
Downloadable Sounds Level 2 region options.

Requirements

Header: Declared in dmdls.h.

DMUS_WAVEDATA

Contains a data chunk for a DLS wave download. The nature of the wave data is defined by the **WaveformatEx** member of the **DMUS_WAVE** structure.

```
typedef struct {
    ULONG cbSize;
    BYTE byData[ ];
} DMUS_WAVEDATA;
```

Members

cbSize
Size of data.

byData[ ]
PCM wave data.

Requirements

Header: Declared in dmdls.h.

DMUS_WAVEDL

Contains information about wave data downloaded to the synthesizer.

```
typedef struct _DMUS_WAVEDL {
    ULONG  cbWaveData;
}   DMUS_WAVEDL, *LPDMUS_WAVEDL;
```

Members

cbWaveData
Number of bytes of wave data.

Requirements

Header: Declared in dmdls.h.

DirectMusic File Format

This section describes the format of files created in DirectMusic Producer and read by DirectMusic when **IDirectMusicLoader8::GetObject** or **IDirectMusicLoader8::LoadObjectFromFile** is called. Most applications don't parse these files directly. This format information is included for developers of music-authoring applications or DirectMusic plug-ins who want to be able to save data in a compatible format or load data into their own objects.

DirectMusic data is stored in the resource interchange file format (RIFF). The following topics contain general information about RIFF files.

- About RIFF
- RIFF Notation

The following topics describe how DirectMusic data is organized in RIFF chunks.

- Common Chunks
- Audiopath Form
- Band Form
- Chord Map Form
- Container Form
- DirectSound Buffer Configuration Form
- Effects Form
- Reference List
- Script Form
- Segment Form
- Song Form
- Style Form
- Tool Form
- Toolgraph Form
- Track Form
- Wave Header Chunk

The following section is a reference to structures used to contain data in RIFF chunks.

- DirectMusic File Structures

About RIFF

The basic building block of a RIFF file is a *chunk*. A chunk is a logical unit of data. Each chunk contains the following fields:

- A four-character code (FOURCC) specifying the chunk identifier. Conventionally, this is uppercase for registered chunk types, and lowercase otherwise.
- A DWORD value specifying the size of the data member in the chunk.
- The data.

A chunk contained in another chunk is a *subchunk*. The only chunks allowed to contain subchunks are those with a chunk identifier of RIFF or LIST.

The first chunk in a file must be identified as RIFF. All other chunks in the file are subchunks of this chunk. RIFF chunks are also called forms.

A LIST chunk is a grouping of subchunks. Some of these subchunks might appear multiple times, but a LIST is not an array. The terminology can be confusing. You might expect the chunk labeled <part-list>, for example, to be a list of musical parts. In fact, it is a list of the elements of a "part" chunk, which describes a single part.

RIFF chunks include an additional field in the first 4 bytes of the data field. This additional field provides the form type of the chunk. The form type is a four-character code identifying the format of the data stored in the file. For example, DirectMusic styles have the form type DMST.

LIST chunks also include an additional field in the first 4 bytes of the data field. This field contains the *list type* of the field. The list type is a four-character code identifying the contents of the list. For example, DirectMusic styles have a LIST chunk with a list type of "part" that contains data pertaining to a particular part (instrument track) in the performance.

Note

Every four-character code used in DirectMusic files has a corresponding macro in Dmusicf.h. For example, the **FOURCC** for DMST is returned by the **DMUS_FOURCC_STYLE_FORM** macro.

For more information on RIFF files in general, see *Resource Interchange File Format Services* in the Platform SDK documentation.

RIFF Notation

The descriptions of DirectMusic files in the following sections use a subset of the conventional notation for RIFF files. The principal parts of this notation are shown in the following table:

Notation	Description
<element>	File element labeled "element", or of type **element**.
[<element>]	Optional file element.
<element>...	One or more copies of the specified element.
[<element>]...	Zero or more copies of the specified element.
name, 'name', NAME, or 'NAME'	**FOURCC** identifier of a form type, list type, or chunk.
// Comment	Comment.

Labels are used only in the notation, not in the files themselves. The label <cheh-ck> refers to a chunk with a unique **FOURCC** identifier and format. Wherever a chunk of this kind occurs in the notation, the same label is used.

The data or subelements associated with a label are shown as in the following example:

```
<cheh-ck> -> cheh( <DMUS_IO_CHORDENTRY> )
```

This notation indicates that the chunk labeled <cheh-ck> consists of the **FOURCC** identifier "cheh" followed by a **DMUS_IO_CHORDENTRY** structure.

Note

The data in every chunk is preceded by a **DWORD** showing the size of the data. This element is not shown in the notation.

The next example shows a list element, consisting of the **FOURCC** "LIST" followed by the list identifier "cmap" and one or more elements labeled <choe-list>. The <choe-list> element would be expanded elsewhere.

```
<cmap-list> -> LIST( 'cmap'
   <choe-list>... )
```

Common Chunks

The following chunks occur in various list chunks and forms.

<guid-ck>

This is the GUID identifier of the element.

```
<guid-ck> -> guid( <GUID> )
```

<vers-ck>

This chunk contains version information for the element.

```
<vers-ck> -> vers( <DMUS_IO_VERSION> )
```

<UNFO-list>

The UNFO chunk is like a standard RIFF INFO list, except that it uses Unicode characters. INFO and UNFO lists consist of various chunks that contain null-terminated strings.

```
<UNFO-list> -> LIST( 'UNFO'
   <unfo-text-ck>...
   )
```

See Also

Reference List

Audiopath Form

The following notation shows the organization of the audiopath form.

```
RIFF( 'DMAP'
   [<guid-ck>]       // GUID for path
   [<vers-ck>]       // Version information
```

```
    [<UNFO-list>]     // Name, author, copyright, comments
    [<DMTG-form>]     // Toolgraph
    [<pcsl-list>]     // Port configurations
    [<dbfl-list>]...  // DirectSound buffer descriptors
)
```

All elements are optional.

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see Common Chunks.

<DMTG-form>

See Toolgraph Form.

<pcsl-list>

Information about the configuration of ports is stored in this list.

```
<pcsl-list> -> LIST( 'pcsl'
    <pcfl-list>...
    )
```

The port configuration list consists of an array of lists in the following format:

```
LIST( 'pcfl'
    <pcfh-ck>         // Header chunk
    <pprh-ck>         // Port parameters used to create the port
    [<dbfl-list>]...  // DirectSound buffer descriptors
    [<pchl-list>]     // Pchannel-to-buffer assignments
    )
```

The port configuration list consists of several chunks, starting with headers describing the port configuration and parameters:

```
<pcfh-ck> -> pcfh( <DMUS_IO_PORTCONFIG_HEADER> )

<pprh-ck> -> pprh( <DMUS_PORTPARAMS8> )
```

The optional buffer descriptors are each contained in a chunk with the following format.

```
<dbfl-list> -> LIST( 'dbfl'
    <ddah-ck>       // Buffer attributes header
    [<DSBC-form>]   // Buffer configuration
)
```

This buffer description list begins with a header describing buffer attributes.

```
<ddah-ck> -> ( 'ddah' < DMUS_IO_BUFFER_ATTRIBUTES_HEADER > )
```

The header is followed by an optional DirectSound Buffer Configuration Form. This is not required for standard buffer types.

The final chunk in the port configuration list is a list containing one or more assignments of performance channels to buffers.

```
pchl-list -> LIST( 'pchl'
    <pchh-ck>...
)
```

This list consists of an array of chunks, each of which describes one assignment of channels to buffers.

```
<pchh-ck> -> pchh(
    <DMUS_IO_PCHANNELTOBUFFER_HEADER>
    <GUID>... // Array of GUIDs specifying the buffers
)
```

<dbfl-list>

The last part of the audiopath form consists of optional buffer descriptors identical in format to those in the port configuration list, <pcsl-list>.

Band Form

The following notation shows the format of the top-level chunk, or form, of a band file. Band forms can also be contained in other chunks.

```
RIFF( 'DMBD'
    [<guid-ck>]     // GUID for band
    [<vers-ck>]     // Optional version information
    [<UNFO-list>]   // Name, author, copyright information, comments
    <lbil-list>     // Instruments
    )
```

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see Common Chunks.

<lbil-list>

The data is contained in an array of lists.

```
<lbil-list> -> LIST( 'lbil'
    <lbin-list>...
    )
```

Each instrument is described in a list that has the following format:

```
<lbin-list> -> LIST( 'lbin'
   <bins-ck>
   [<DMRF-list>]
   )
```

Within the instrument list, the following chunk contains a header describing the instrument:

```
<bins-ck> -> bins( <DMUS_IO_INSTRUMENT> )
```

The instrument list can also contain <DMRF-list>, which in this case is a reference to a DLS file. See Reference List.

Chord Map Form

The following notation shows the organization of the top-level chunk, or form, of a chordmap file:

```
RIFF( 'DMPR'
   <perh-ck>       // Chord map header chunk
   [ <guid-ck> ]   // GUID chunk
   [ <vers-ck> ]   // Version chunk
   [ <UNFO-list> ] // UNFO list
   <chdt-ck>       // Chord data chunk
   <chpl-list>     // Chord palette
   <cmap-list>     // Chord graph
   <spsq-list>     // Signpost list
)
```

<perh-ck>

This is the basic header information for a chordmap.

```
<perh-ck> -> perh( <DMUS_IO_CHORDMAP> )
```

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see Common Chunks.

<chdt-ck>

```
<chdt-ck> -> chdt(
   <WORD>    // Size of DMUS_IO_CHORDMAP_SUBCHORD
   <DMUS_IO_CHORDMAP_SUBCHORD>...
   )
```

The <chdt-ck> chunk contains a **WORD** indicating the number of bytes per subchord followed by an array of unique subchords. The subchord identifiers referred to in other parts of this file all correspond directly to an index into this array.

<chpl-list>

```
<chpl-list> -> LIST( 'chpl'
   <chrd-list>...
   )
```

This list contains the chord palette. There must be exactly 24 items in this list.

```
<chrd-list> -> LIST( 'chrd'
   <UNAM-ck>    // Chord name
   <sbcn-ck>    // Subchord indexes
   )
```

This list contains the basic chord information. This information is simply the chord's name and a list of identifiers for the subchords it comprises.

```
<UNAM-ck> -> UNAM ( <WCHAR>... )
```

The UNAM chunk stores the name of the chord.

```
<sbcn-ck> -> sbcn( <WORD>... )
```

The "sbcn" chunk contains one or more subchord identifiers. These correspond directly to an index into the array found in <chdt-ck>. A maximum of four chords is supported.

<cmap-list>

This list contains the entire chord connection graph for the chordmap. The bulk of the data for the chordmap resides in this chunk.

```
<cmap-list>  ->  LIST( 'cmap'
   <choe-list>...
   )
```

Each list contains data for a single entry in the chord graph, along with pointers to all the chords that can occur next in the chord graph.

```
<choe-list> -> LIST( 'choe'
   <cheh-ck>      // Chord entry data
   <chrd-list>    // Chord data; see above.
   <ncsq-ck>      // Next chord list
   )

<cheh-ck> -> cheh( <DMUS_IO_CHORDENTRY> )
```

This is the chord entry header. The identifier in the structure is the identifier for the chord connection graph, not a subchord identifier.

```
<ncsq-ck> -> ncsq (
   <WORD>        // Size of DMUS_IO_NEXTCHORD
   <DMUS_IO_NEXTCHORD>...
   )
```

The "ncsq" chunk contains data that connects one chord in the connection graph to another. Each chord in the connection graph is represented by a 16-bit identifier.

<spsq-list>

This chunk contains data for each of the signposts.

```
<spsq-list> -> LIST( 'spsq'
   <spst-list>...
   )
```

The <spst-list> contains data for a single signpost, consisting of a header, chord information, and optional cadence chords.

```
<spst-list>  ->  LIST( 'spst'
   <spsh-ck>
   <chrd-list>     // Chord data
   [ <cade-list> ] // Cadence chords
   )
```

The <spsh-ck> contains the signpost data.

```
<spsh-ck> -> spsh( <DMUS_IO_CHORDMAP_SIGNPOST> )
```

For <chrd-list>, see <chpl-list>, described previously.

The <cade-list> chunk contains the chord information for cadence chords. Support for up to two cadence chords in this list is provided. Any additional chords are ignored.

```
<cade-list> -> LIST( 'cade'
   <chrd-list>...
   )
```

Container Form

The container form is a chunk that contains other chunks such as segment or style forms. It can be contained within a segment or script file. It is organized as follows:

```
RIFF ( 'DMCN'
   <conh-ck>       // Container header chunk
   [<guid-ck>]     // GUID for container
```

```
    [<vers-ck>]     // Optional version information
    [<UNFO-list>]   // Name, author, copyright information, comments
    <cosl-list>     // List of objects.
)
```

<conh-ck>

This chunk contains a header structure.

```
<conh-ck> -> 'conh' ( <DMUS_IO_CONTAINER_HEADER> )
```

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see Common Chunks.

<cosl-list>

The final element of the container form is an array of chunks that describe the objects in the container.

```
<cosl-list> -> LIST ( 'cosl'
    <cobl-list>...
    )
```

Each object is stored in the following format:

```
<cobl-list> -> LIST( 'cobl'
    [<coba-ck>]
    <cobh-ck>       // Required header
    [<data> ] or [ DMRF-list ]
```

The first element is an alias, or alternative name by which this object is known within the container.

```
<coba-ck> -> coba( <WCHAR>... )  // Null-terminated string
```

The second element header for the object.

```
<cobh-ck> -> cobh ( <DMUS_IO_CONTAINED_OBJECT_HEADER> )
```

The header is normally followed by object data of the type specified in <cobh-ck>. This can be in the form of a RIFF chunk such as a Segment Form or a Style Form. If it is a DMRF-list, it is a reference to the object. For more information on DMRF, see Reference List.

DirectSound Buffer Configuration Form

The following notation shows the organization of a chunk containing information about a DirectSound buffer configuration.

```
RIFF ( 'DSBC'
    [<guid-ck>]     // GUID identifier for this buffer configuration
    [<vers-ck>]     // Optional version information
    [<UNFO-list>]   // Name, author, copyright information, comments
    <dsbd-ck>       // DirectSound buffer descriptor
    [<bsid-ck>]     // Bus identifiers
    [<ds3d-ck>]     // 3-D parameters
    [<fxls-list>]   // Effect descriptors
    )
```

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see Common Chunks.

<dsbd-ck>

The DirectSound buffer descriptor is organized as follows:

```
<dsbd-ck> -> 'dsbd' (
    <DSOUND_IO_DSBUFFERDESC>
    )
```

<bsid-ck>

The bus identifiers are stored in the following chunk.

```
<bsid-ck> -> 'bsid' ( <DSOUND_IO_DSBUSID> )
```

The **DSOUND_IO_DSBUSID** structure is an array of bytes whose size is specified by the chunk size.

<ds3d-ck>

The 3-D parameters of the buffer are stored in the following chunk.

```
<ds3d-ck> -> 'ds3d'( <DSOUND_IO_3D> )
```

<fxls-list>

The next list contains information about DMOs associated with the buffer.

```
<fxls-list> -> LIST ( 'fxls'
    <DSFX-form>...
    )
```

Each DMO is stored in an Effects Form.

Effects Form

The effects form describes a DMO effect.

```
RIFF ( 'DSFX'
   <fxhr-ck>
   [<data-ck>]
)
```

<fxhr-ck>

The header chunk describes the effect.

```
<fxhr-ck> -> fxhr ( <DSOUND_IO_DXDMO_HEADER> )
```

<data-ck>

The data chunk is an optional set of values for the effect parameters in the format expected by the DMO.

```
<data-ck> -> data ( <DSOUND_IO_DXDMO_DATA> )
```

Reference List

The reference list chunk contains information about a reference to an object in another file. For example, a band object might contain a reference to a DLS collection in a separate file. This subchunk is used in many different chunks.

The notation for a reference list is as follows:

```
<DMRF-list> ->LIST( 'DMRF'
   <refh-ck>       // Reference header
   [<guid-ck>]     // Object GUID
   [<date-ck>]     // File date
   [<name-ck>]     // Name
   [<file-ck>]     // File name
   [<catg-ck>]     // Category name
   [<vers-ck>]     // Version information
)
```

<refh-ck>

The data begins with a header that includes information about the object being referred to:

```
<refh-ck> -> refh( <DMUS_IO_REFERENCE> )
```

All other chunks are optional.

<guid-ck>

See Common Chunks.

<date-ck>

The date chunk contains a date in a **FILETIME** structure.

```
<date-ck> -> date( <FILETIME> )
```

<name-ck>, <file-ck>, <catg-ck>

The name, file name, and category name are null-terminated strings.

```
<name-ck> -> name( <WCHAR>... )
<file-ck> -> file( <WCHAR>... )  // Null-terminated string
<catg-ck> -> catg( <WCHAR>... )  // Null-terminated string
```

<vers-ck>

See Common Chunks.

Script Form

This section describes the organization of a chunk containing information about a DirectMusic script.

```
RIFF ( 'DMSC'
    <schd-ck>       // Script header chunk
    [<guid-ck>]     // GUID for script
    [<vers-ck>]     // Optional version information
    [<UNFO-list>]   // Name, author, copyright information, comments
    <scve-ck>       // Version of DirectMusic
    <DMCN-form>     // Container of content referenced by the script
    <scla-ck>       // Scripting language
    <scsr-ck> or <DMRF> // Source code
)
```

<schd-ck>

The header chunk contains flags in a **DMUS_IO_SCRIPT_HEADER** structure.

```
<schd-ck> -> schd( <DMUS_IO_SCRIPT_HEADER> )
```

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see Common Chunks.

<scve-ck>

This chunk describes the version of DirectMusic against which the script was authored.

```
<scve-ck> -> scve( <DMUS_IO_VERSION> )
```

<DMCN-form>

For information on this chunk, see Container Form.

<scla-ck>

This chunk consists of a null-terminated string describing the scripting language.

```
<scla-ck> -> scla( <WCHAR>... )
```

<scsr-ck> or <DMRF>

The final chunk can contain the source code as a null-terminated array of **WCHAR** values.

```
<scsr-ck> -> scsr( <WCHAR>... )
```

Alternatively, the final chunk can be a Reference List chunk pointing to a text file containing the script code. The **guidClassID** member of the **DMUS_IO_REFERENCE**) must be GUID_NULL, because this text file is not a DirectMusic object.

Segment Form

The following notation shows the organization of the top-level chunk of a segment file. This form can also be contained within a Container Form.

```
RIFF( 'DMSG'
    <segh-ck>       // Segment header chunk
    [<guid-ck>]     // GUID for the segment
    [<vers-ck>]     // Optional version information
    [<UNFO-list>]   // Name, author, copyright information, comments
    [<DMCN-form>]   // Optional container of objects embedded in file
    <trkl-list>     // Tracks
    [<DMTG-form>]   // Optional toolgraph
    [<DMAP-form>]   // Optional audiopath
    )
```

<segh-ck>

This chunk contains the basic header information for a segment.

```
<segh-ck> -> segh( <DMUS_IO_SEGMENT_HEADER> )
```

<guid-ck>, <vers-ck>, <UNFO-list>

See Common Chunks.

<DMCN-form>

See Container Form.

<trkl-list>

This is the track list. Each track is encapsulated in a Track Form.

```
<trkl-list> -> LIST( 'trkl'
    <DMTK-form>...
    )
```

<DMTG-form>

See Toolgraph Form.

<DMAP-form>

See Audiopath Form.

Song Form

Not implemented in DirectX 8.0.

The following notation shows the organization of the top-level chunk of a song file. This form can also be contained within a Container Form.

```
RIFF( 'DMSO'
    <sngh-ck>       // Song header chunk
    [<guid-ck>]     // GUID for song
    [<vers-ck>]     // Optional version info
    [<UNFO-list>]   // Name, author, copyright information, comments
    [<DMCN-form>]   // Container of objects embedded in file
    <segl-list>     // Segments
    [<tlgl-list>]   // Toolgraphs
    [<DMAP-form>]   // Audiopath shared by all segments
    <srsl-list>     // Segment references
)
```

<sngh-ck>

This chunk contains the basic header information for a song.

```
<sngh-ck> -> sngh ( <DMUS_IO_SONG_HEADER> )
```

<guid-ck>, <vers-ck>, <UNFO-list>

See Common Chunks.

<DMCN-form>

See Container Form.

<segl-list>

```
<segl-list> -> LIST( 'segl'
    <ssgl-list>...
    )
```

The song segments list contains an array of segments, each of which is described in the following list.

```
<ssgl-list> -> LIST( 'ssgl'
    [<DMSG-form>]
    [<DMRF-list>]
    )
```

This list consists of either a Segment Form or a Reference List.

<tlgl-list>

```
<tlgl-list> -> LIST( 'tlgl'
    <DMTG-form>...
    )
```

The toolgraph list consists of an array of Toolgraph Forms.

<DMAP-form>

This chunk describes audiopaths shared by all segments in the song. See Audiopath Form.

<srsl-list>

The last chunk in the song form is an array of segment reference lists that are organized as follows:

```
<srsl-list> -> LIST( 'srsl'
   <sgrl-list>...
   )

<sgrl-list> -> LIST( 'sgrl'
   <sgrh-ck>       // Segment reference header
   <segh-ck>       // Segment header
   [<UNFO-list>]   // Name, author, copyright, comments
   [<strh-ck>]     // Segment transition chunk
   [<trsl-list>]   // Track references
   )
```

This list starts with a segment reference header.

```
<sgrh-ck> -> ( 'sgrh' <DMUS_IO_SEGREF_HEADER> )
```

Next is a segment header. For the syntax of the <segh-ck> chunk, see Segment Form.

For <UNFO-list>, see Common Chunks.

The segment transition chunk specifies valid transitions from other segments to this segment. It is organized as follows:

```
<strh-ck> -> ( strh
   <DMUS_IO_TRANSITION_DEF>    // Default transition
   <DMUS_IO_TRANSITION_DEF>... // Additional transitions
   )
```

The last part of the segment reference list chunk contains track references that are used to create a segment from tracks in multiple segments:

```
<trsl-list> -> LIST( 'trsl'
   <tkrl-list>...
   )
```

This list consists in turn of an array of track reference lists, each of which is organized as follows:

```
<tkrl-list> -> ( 'tkrl'
   <tkrh-ck>       // Track reference header chunk
   <trkh-ck>       // Track header chunk
   [<trkx-ck>]     // Track flags
   )
```

This list consists of three chunks. The first consists of a header structure that identifies the segment containing the track:

```
tkrh-ck -> tkrh( <DMUS_IO_TRACKREF_HEADER> )
```

For the other two chunks of the track reference list, see Track Form.

Style Form

The following notation shows the organization of the top-level chunk of a style file. This form can also be contained within a Container Form.

```
RIFF( 'DMST'
    <styh-ck>       // Style header chunk
    <guid-ck>       // Unique identifier
    [<UNFO-list>]   // Name, author, copyright information, comments
    [<vers-ck>]     // Version chunk
    <part-list>...  // Array of parts in the style, used by patterns
    <pttn-list>...  // Array of patterns in the style
    <DMBD-form>...  // List of bands in the style
    [<prrf-list>]   // List of chordmap references in the style
)
```

<styh-ck>

This chunk contains the basic header information for a style.

```
<styh-ck> -> styh( <DMUS_IO_STYLE> )
```

<guid-ck>,<UNFO-list>, <vers-ck>

For information on these three chunks, see Common Chunks.

<part-list>

Each musical part in the style is described in a chunk with the following format.

```
<part-list> -> LIST('part'
    <prth-ck>       // Part header chunk
    [<UNFO-list>]
    [<note-ck>]     // Notes in part
    [<crve-ck>]     // Curves in part
    [<mrkr-ck>]     // Markers in part
    [<rsln-ck>]     // Variation resolutions in part
    [<anpn-ck>]     // Resolution anticipations in part
    )
```

The part list includes a header, an optional UNFO chunk, and a list of elements, as shown in the following notation. (For the UNFO list, see Common Chunks.)

```
<prth-ck> -> prth( <DMUS_IO_STYLEPART> )

<note-ck> -> note(
    < DWORD >  // Size of DMUS_IO_STYLENOTE
```

```
   < DMUS_IO_STYLENOTE >...
   )

<crve-ck> -> crve(
   < DWORD >  // Size of DMUS_IO_STYLECURVE
   < DMUS_IO_STYLECURVE >...
   )

<mrkr-ck> -> mrkr(
   < DWORD > // Size of DMUS_IO_STYLEMARKER
   < DMUS_IO_STYLEMARKER >...
   )

<rsln-ck> -> rsln(
   < DWORD > // Size of DMUS_IO_STYLERESOLUTION
   < DMUS_IO_STYLERESOLUTION >...
   )

<anpn-ck> -> anpn(
   < DWORD > Size of DMUS_IO_STYLE_ANTICIPATION
   < DMUS_IO_STYLE_ANTICIPATION >...
   )
```

<pttn-list>

Each pattern is described in a chunk with the following format.

```
<pttn-list> -> LIST( 'pttn'
   <ptnh-ck>       // Pattern header chunk
   <rhtm-ck>       // List of rhythms for chord matching
   <swpt-ck>       // List of pattern switch points
   [<UNFO-list>]
   [<mtfs-ck>]     // Motif settings chunk
   [<DMBD-form>]   // Band to be associated with the pattern
                   // (for motifs)
   <pref-list>...  // Array of part reference IDs
   )
```

The first chunk of the pattern list is a header:

```
<ptnh-ck> -> ptnh( <DMUS_IO_PATTERN> )
```

The second chunk is a rhythm list:

```
<rhtm-ck> -> rhtm( <DWORD>... )
```

This chunk consists of an array of **DWORD**s, one for each measure, giving the rhythm pattern. For information on the arrangement of the bits, see the **dwRhythmPattern** member of **DMUS_RHYTHM_PARAM**.

The third chunk of the pattern list is the switch point chunk:

```
<swpt-ck> -> swpt( <DWORD>... )
```

The switch point chunk consists of an array of **DWORD**s, one for each measure, giving valid points where the pattern can start.

For the optional UNFO list, see Common Chunks.

The next chunk of the pattern list describes the motif settings:

```
<mtfs-ck> -> mtfs( <DMUS_IO_MOTIFSETTINGS> )
```

For the <DMBD-form> chunk of the pattern list, see Band Form.

The last chunk of the pattern list is a part reference list.

```
<pref-list> -> LIST( 'pref'
   <prfc-ck>   // Part reference chunk
   )
```

The only element is a part reference.

```
<prfc-ck> -> prfc( <DMUS_IO_PARTREF> )
```

<DMBD-form>

The next chunk in the style form is a Band Form.

<prrf-list>

The final chunk contains an array of chordmap references:

```
<prrf-list> -> LIST( 'prrf'
   <DMRF-list>...
   )
```

For more information on <DMRF-list>, see Reference List.

Tool Form

The tool form contains information about tools. Tools can be embedded in a Toolgraph Form or stored as separate files.

```
<DMTL-form> -> RIFF( 'DMTL'
   <tolh-ck>
   [<data>]       // Tool data
```

```
)
```

<tolh-ck>

This is the tool header chunk.

```
<tolh-ck> -> tolh( <DMUS_IO_TOOL_HEADER> )
```

<data>

The <data> element is a chunk of the type identified in the **DMUS_IO_TOOL_HEADER** structure. The format of this chunk depends on the definition of the tool. It can be a list or a normal chunk.

Toolgraph Form

A toolgraph chunk can occur either as a top-level form or as a subchunk of a Segment Form or Container Form.

```
RIFF( 'DMTG'
    [<guid-ck>]     // GUID for toolgraph
    [<vers-ck>]     // Version information
    [<UNFO-list>]   // Name, author, copyright information, comments
    <toll-list>     // List of tools
)
```

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see Common Chunks.

<toll-list>

The main and only required part of the toolgraph chunk is the tool list, which consists of an array of tool forms:

```
<toll-list> -> LIST( 'toll'
    <DMTL-form>...
    )
```

For more information on the <DMTL-form> chunk, see Tool Form.

Track Form

The track form contains information about a single track. It can be embedded in a Segment Form or stored in its own file.

```
<DMTK-form> -> RIFF( 'DMTK'
```

```
    <trkh-ck>
    [<trkx-ck>]     // Optional track flags
    [<guid-ck>]     // GUID for track object instance
    [<vers-ck>]     // Version information
    [<UNFO-list>]   // Name, author, copyright information, comments
    [<data>]        // Track data
)
```

<trkh-ck>

The first chunk contains the basic header information for a track.

```
<trkh-ck> -> trkh( <DMUS_IO_TRACK_HEADER> )
```

<trkx-ck>

This optional chunk contains flags for the track.

```
<trkx-ck> -> trkx( <DMUS_IO_TRACK_EXTRAS_HEADER> )
```

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see Common Chunks.

<data>

The last element in the track form is the data for the track. The chunk type used for the data is identified in the **DMUS_IO_TRACK_HEADER** structure. The following standard track chunks are defined:

- Band Track Form
- Chord Track List
- Chord Map Track List
- Command Track Chunk
- Lyrics Track List
- Marker Track List
- Melody Formulation Track List
- Mute Track Chunk
- Parameter Control Track List
- Pattern Track Form
- Script Track List
- Segment Trigger Track List
- Sequence Track Chunk

- Signpost Track Chunk
- Style Track List
- Sysex Track Chunk
- Tempo Track Chunk
- Time Signature Track List
- Wave Track List

Band Track Form

The band track form can be a top-level form but is also found as the data part of a Track Form. It is organized as follows:

```
RIFF( 'DMBT'
   [<bdth-ck>]    // Band track header
   [<guid-ck>]    // GUID for band track
   [<vers-ck>]    // Version information
   [<UNFO-list>]  // Name, author, copyright information, comments
   <lbdl-list>    // List of band lists
   )
```

<bdth-ck>

This optional chunk contains header information for a band track. The only data in the structure is a flag for automatic downloading.

```
<bnth-ck> -> bdth( <DMUS_IO_BAND_TRACK_HEADER> )
```

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see Common Chunks.

<lbdl-list>

The last chunk contains one or more bands.

```
<lbdl-list> -> LIST( 'lbdl'
   <lbnd-list>...
   )
```

Each band is encapsulated in a list of the following type.

```
<lbnd-list> -> LIST( 'lbnd'
   <bdih-ck> or <bd2h-ck>
   <DMBD-form>
   )
```

The band list begins with a header. In older files, this is <bdih-ck>; newer content uses <bd2h-ck>.

```
<bdih-ck> -> ( <DMUS_IO_BAND_ITEM_HEADER> )

<bd2h-ck> -> ( <DMUS_IO_BAND_ITEM_HEADER2> )
```

The header is followed by a Band Form containing information about the instruments in the band.

Chord Track List

The chord track list contains chord data for a Track Form. It is organized as follows:

```
<cord-list> -> LIST( 'cord'
   <crdh-ck>      // Header
   <crdb-ck>      // Chord body chunk
   )
```

<crdh-ck>

```
<crdh-ck> -> crdh ( <DWORD> )
```

The header is a **DWORD** containing the chord root in the upper 8 bits and the scale in the lower 24 bits. For an explanation of what these bits represent, see **DMUS_IO_SUBCHORD**.

The body of data for the chord track list consists of information about a chord change and the component subchords:

```
<crdb-ck> -> crdb(
   <DWORD>          // Size of DMUS_IO_CHORD
   <DMUS_IO_CHORD>
   <DWORD>          // Number of subchords
   <DWORD>          // Size of DMUS_IO_SUBCHORD
   <DMUS_IO_SUBCHORD>...
   )
```

Chord Map Track List

The chordmap track list contains data for a Track Form. It is organized as follows:

```
<pftr-list> -> LIST('pftr'
   <pfrf-list>...
   )
```

The data consists of one or more lists containing time stamps and references to chordmaps:

```
<pfrf-list> -> LIST('pfrf'
   <stmp-ck>
   <DMRF-list>
   )
```

The notation for the time stamp chunk is as follows:

```
<stmp-ck> -> stmp( <DWORD> )
```

For information on <DMRF-list>, see Reference List.

Command Track Chunk

The command track chunk contains data for a Track Form. It is organized as follows:

```
<cmnd-ck> -> cmnd(
   <DWORD>    //Size of DMUS_IO_COMMAND
   <DMUS_IO_COMMAND>...
   )
```

Lyrics Track List

The lyrics track list contains data for a Track Form.

```
<lyrt-list> -> LIST( 'lyrt'
   <lyrl-list>
   )
```

The sole chunk in the lyrics track list is another list containing an array of lyrics events:

```
<lyrl-list> -> LIST(
   <lyre-list>...
   )
```

Each lyrics event is stored in another list, as follows:

```
<lyre-list> -> LIST(
   <lyrh-ck>        // Event header chunk
   <lyrn-ck>        // Notification text
   )
```

The first chunk is a header:

```
<lyrh-ck> -> lyrh( <DMUS_IO_LYRICSTRACK_EVENTHEADER> )
```

The second chunk contains the text associated with the event, in a null-terminated string:

```
<lyrn-ck> -> lyrn( <WCHAR>... )
```

Marker Track List

The marker track list contains data for a Track Form. It is organized as follows:

```
<mark-list> -> LIST ( 'MARK'
   [<vals-ck>]
   [<play-ck>]
)
```

The first element in the list is an array of chunks defining valid start points:

```
<vals-ck> -> vals(
   DWORD    // size of DMUS_IO_VALID_START
   <DMUS_IO_VALID_START>...
   )
```

The second element is an array of chunks defining valid play points.

```
<play-ck> -> play(
   DWORD    // size of DMUS_IO_PLAYMARKER
   <DMUS_IO_PLAYMARKER>...
   )
```

Melody Formulation Track List

Not implemented in DirectX 8.0.

The melody formulation track list contains data for a Track Form. It is organized as follows:

```
<mfrm-list> -> LIST( 'mfrm'
   <mlfh-ck>    // Melody formulation header
   <mlfb-ck>    // Melody formulation body
   )
```

The melody formulation header chunk looks like this:

```
<mlfb-ck> -> mlfb( <DMUS_IO_MELFORM> )
```

The body of the data is stored in the following chunk:

```
<mlfb-ck> -> mlfb(
   <DWORD>    //size of DMUS_IO_MELODY_FRAGMENT
   <DMUS_IO_MELODY_FRAGMENT>...
   )
```

DMUS_IO_MELODY_FRAGMENT is defined as equivalent to DMUS_MELODY_FRAGMENT.

Mute Track Chunk

The mute track chunk contains data for a Track Form. It is organized as follows:

```
<mute-ck> -> mute(
   <DWORD>    //Size of DMUS_IO_MUTE
   <DMUS_IO_MUTE>...
   )
```

Parameter Control Track List

The parameter control track list contains data for a Track Form.

```
<prmt-list> -> LIST( 'prmt'
   <prol-list>...
   )
```

The list contains an array of lists, each of which describes an object.

```
<prol-list> -> LIST( 'proh'
   <proh-ck>       // Object header chunk
   <prpl-list>...  // Array of parameters
   )
```

The first chunk in the object list is a header.

```
<proh-ck> -> proh( <DMUS_IO_PARAMCONTROLTRACK_OBJECTHEADER> )
```

The second chunk contains parameter lists, organized as shown in the following notation:

```
<prpl-list> -> LIST( 'prpl'
   <prph-ck>       // Parameter header
   <prcc-ck>       // Array of curves
   )
```

The header of the parameter list is described as follows:

```
<prph-ck> -> prph( <DMUS_IO_PARAMCONTROLTRACK_PARAMHEADER> )
```

The array of curves is described in the following chunk.

```
<prcc-ck> -> prcc(
   <DWORD>   // Sizeof DMUS_IO_PARAMCONTROLTRACK_CURVEINFO
  <DMUS_IO_PARAMCONTROLTRACK_CURVEINFO>... // Curves, sorted in order of mtTime
   )
```

Pattern Track Form

The pattern track form can be a top-level form but is also found as data for a Track Form. It is organized as follows:

```
RIFF( 'DMPT'
   <styh-ck>      // Style header chunk
   <pttn-list>    // The pattern
   )
```

For information on <styh-ck> and <pttn-list>, see Style Form.

Script Track List

The script track list contains data for a Track Form.

```
<scrt-list> -> LIST( 'scrt'
   <scrl-list>
   )
```

The list contains another list that contains an array of script events.

```
<scrl-list> -> LIST( 'scrl'
   <scre-list>...
   )
```

Each <scre-list> chunk describes an event as follows:

```
<scre-list> -> LIST( 'scre'
   <scrh-ck>       // Event header chunk
   <DMRF>           // Reference
   <scrn-ck>       // Routine name
   )
```

Each script track event begins with a header:

```
<scrh-ck> -> 'scrh'( <DMUS_IO_SCRIPTTRACK_EVENTHEADER> )
```

For information on the DMRF chunk, see Reference List.

The last chunk of the script track event is the null-terminated name of a routine:

```
<scrn-ck> -> 'scrn'( <WCHAR>... )
```

Segment Trigger Track List

The segment trigger track list contains data for a Track Form. It is organized as follows:

```
<SEGT-list> -> LIST( 'segt'
```

```
[<sgth-ck>]    // Segment track header
<lsgl-list>    // List of segment lists
)
```

The first chunk is the track header:

```
<sgth-ck> -> 'sgth' ( <DMUS_IO_SEGMENT_TRACK_HEADER> )
```

The next chunk is a list containing an array of segments:

```
<lsgl-list> -> LIST( 'lsgl'
   <lseg-list>...
   )
```

Each "lseg" list describes a single segment item:

```
<lseg-list> -> LIST( 'lseg'
   <sgih-ck>
   <DMRF-list>
   [<snam-ck>]    // Motif name
   )
```

The first chunk of the segment item is a header:

```
<sgih-ck> -> ( <DMUS_IO_SEGMENT_ITEM_HEADER> )
```

This is followed by a reference to a segment file or a style file. It is a reference to a style if the DMUS_SEGMENTTRACKF_MOTIF flag is present in the item header. For more information, see Reference List.

The last chunk of the segment item contains the null-terminated name of a motif, if the DMUS_SEGMENTTRACKF_MOTIF flag is present in the item header.

```
<snam-ck> -> ( <WCHAR>... )
```

Sequence Track Chunk

The sequence track chunk contains data for a Track Form. It is organized as follows:

```
<seqt> -> seqt(
   <evtl-ck>
   <curl-ck>
   )
```

The sequence track chunk can contain two chunks, one for sequence items and one for curve items:

```
<evtl-ck> -> evtl(
   <DWORD>    // Size of DMUS_IO_SEQ_ITEM
   <DMUS_IO_SEQ_ITEM>...
```

```
)

<curl-ck> -> curl(
   <DWORD>     // Size of DMUS_IO_CURVE_ITEM
   <DMUS_IO_CURVE_ITEM>...
   )
```

Note

The sequence track chunk does not conform to the convention that only RIFF and LIST chunks can have subchunks.

Signpost Track Chunk

The signpost track chunk contains data for a Track Form. It is organized as follows:

```
<sgnp-list> -> sgnp(
   <DWORD>    // Size of DMUS_IO_SIGNPOST
   <DMUS_IO_SIGNPOST>...
   )
```

Style Track List

The style track list contains data for a Track Form. It is organized as follows:

```
<sttr-list> -> LIST('sttr'(
   <strf-list>...
   )
```

The data consists of one or more lists containing time stamps and references to styles:

```
<strf-list> -> LIST('strf'(
   <stmp-ck>
   <DMRF-list>
   )
```

The first chunk contains time stamp data, as follows:

```
<stmp-ck> -> stmp( <DWORD> )
```

For information on <DMRF-list>, see Reference List.

Sysex Track Chunk

The sysex track chunk contains data for a Track Form. It is an array of system exclusive message items, each consisting of a **DMUS_IO_SYSEX_ITEM** structure followed by the number of bytes specified in the **dwSysExLength** member.

```
<syex-ck> -> 'syex' (
   {
   <DMUS_IO_SYSEX_ITEM>
```

```
<BYTE>...    // Data
}...
)
```

Tempo Track Chunk

The tempo track chunk contains data for a Track Form. It is organized as follows:

```
<tetr-ck> -> tetr(
   <DWORD>    // Size of DMUS_IO_TEMPO_ITEM
   <DMUS_IO_TEMPO_ITEM>...
   )
```

Time Signature Track List

The time signature track list contains data for a Track Form. It is organized as follows:

```
<tims-list> -> LIST( 'TIMS'
   <tims-ck> // Time signatures
   )
```

The time signature array is contained in the following chunk:

```
<tims-ck> -> tims (
   <DWORD>  // Size of DMUS_IO_TIMESIGNATURE_ITEM
   <DMUS_IO_TIMESIGNATURE_ITEM>...
   )
```

Wave Track List

The following notation shows the organization of a chunk containing data for a wave track:

```
<wavt-list> -> LIST ( 'wavt'
   <wath-ck>      // Wave track header
   <wavp-list>... // Wave parts
   }
```

<wath-ck>

This chunk contains header information for a wave. It is followed by an array of lists describing wave parts:

```
<wath-ck> -> wath( <DMUS_IO_WAVE_TRACK_HEADER> )
```

<wavp-list>

```
<wavp-list> -> LIST ( 'wavp'
   <waph-ck>       //  Wave part header
   <wavi-list>     //  Wave items
   )
```

The wave part list begins with a header.

```
<waph-ck> -> 'waph' ( <DMUS_IO_WAVE_PART_HEADER> )
```

The second part of the wave part list is an array of wave items:

```
<wavi-list> -> LIST( 'wavi'
   <wave-list>...
   )
```

Each wave item is described in a list chunk as follows:

```
<wave-list> -> LIST( 'wave'
   <waih-ck>       //  Wave item header
   <DMRF-list>     //  Reference to wave object
   )
```

The wave description begins with a header chunk:

```
<waih-ck> -> ( <DMUS_IO_WAVE_ITEM_HEADER> )
```

For more information on <DMRF-list>, see Reference List.

Wave Header Chunk

The wave header chunk is a special chunk added to wave files for DirectMusic. It specifies streaming capabilities.

```
<wavh-ck> -> ( <DMUS_IO_WAVE_HEADER>)
```

DirectMusic File Structures

This section contains reference information for data structures used in DirectMusic files. Most applications do not need to know about these structures because each standard DirectMusic object handles the loading of its own data through its **IPersistStream** interface. The structures are chiefly of interest for music-authoring applications that need to save data in a format compatible with DirectMusic.

The following structures are used in DirectMusic files.

- **DMUS_IO_BAND_ITEM_HEADER**
  DMUS_IO_BAND_ITEM_HEADER2

- **DMUS_IO_BAND_TRACK_HEADER**
- **DMUS_IO_BUFFER_ATTRIBUTES_HEADER**
- **DMUS_IO_CHORD**
- **DMUS_IO_CHORDENTRY**
- **DMUS_IO_CHORDMAP**
- **DMUS_IO_CHORDMAP_SIGNPOST**
- **DMUS_IO_CHORDMAP_SUBCHORD**
- **DMUS_IO_COMMAND**
- **DMUS_IO_CONTAINED_OBJECT_HEADER**
- **DMUS_IO_CONTAINER_HEADER**
- **DMUS_IO_CURVE_ITEM**
- **DMUS_IO_INSTRUMENT**
- **DMUS_IO_LYRICSTRACK_EVENTHEADER**
- **DMUS_IO_MELFORM**
- **DMUS_IO_MOTIFSETTINGS**
- **DMUS_IO_MUTE**
- **DMUS_IO_NEXTCHORD**
- **DMUS_IO_PARAMCONTROLTRACK_CURVEINFO**
- **DMUS_IO_PARAMCONTROLTRACK_OBJECTHEADER**
- **DMUS_IO_PARAMCONTROLTRACK_PARAMHEADER**
- **DMUS_IO_PARTREF**
- **DMUS_IO_PATTERN**
- **DMUS_IO_PCHANNELTOBUFFER_HEADER**
- **DMUS_IO_PLAYMARKER**
- **DMUS_IO_PORTCONFIG_HEADER**
- **DMUS_IO_REFERENCE**
- **DMUS_IO_SCRIPT_HEADER**
- **DMUS_IO_SCRIPTTRACK_EVENTHEADER**
- **DMUS_IO_SEGMENT_HEADER**
- **DMUS_IO_SEGMENT_ITEM_HEADER**
- **DMUS_IO_SEGMENT_TRACK_HEADER**
- **DMUS_IO_SEGREF_HEADER**
- **DMUS_IO_SEQ_ITEM**
- **DMUS_IO_SIGNPOST**
- **DMUS_IO_SONG_HEADER**
- **DMUS_IO_STYLE**
- **DMUS_IO_STYLE_ANTICIPATION**

- **DMUS_IO_STYLECURVE**
- **DMUS_IO_STYLEMARKER**
- **DMUS_IO_STYLENOTE**
- **DMUS_IO_STYLEPART**
- **DMUS_IO_STYLERESOLUTION**
- **DMUS_IO_SUBCHORD**
- **DMUS_IO_SYSEX_ITEM**
- **DMUS_IO_TEMPO_ITEM**
- **DMUS_IO_TIMESIG**
- **DMUS_IO_TIMESIGNATURE_ITEM**
- **DMUS_IO_TOOL_HEADER**
- **DMUS_IO_TRACK_EXTRAS_HEADER**
- **DMUS_IO_TRACK_HEADER**
- **DMUS_IO_TRACKREF_HEADER**
- **DMUS_IO_TRANSITION_DEF**
- **DMUS_IO_VALID_START**
- **DMUS_IO_VERSION**
- **DMUS_IO_WAVE_HEADER**
- **DMUS_IO_WAVE_ITEM_HEADER**
- **DMUS_IO_WAVE_PART_HEADER**
- **DMUS_IO_WAVE_TRACK_HEADER**
- **DSOUND_IO_3D**
- **DSOUND_IO_DSBUFFERDESC**
- **DSOUND_IO_DSBUSID**
- **DSOUND_IO_DXDMO_DATA**
- **DSOUND_IO_DXDMO_HEADER**

See Also

DirectMusic File Format

DMUS_IO_BAND_ITEM_HEADER

Contains information about a band change. Used in the Band Track Form of older files. It has been superseded by **DMUS_IO_BAND_ITEM_HEADER2**.

```
typedef struct _DMUS_IO_BAND_ITEM_HEADER {
    MUSIC_TIME lBandTime;
} DMUS_IO_BAND_ITEM_HEADER;
```

Members

lBandTime
Time of the band change.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_BAND_ITEM_HEADER2

Contains information about a band change. Used in the Band Track Form.

```
typedef struct _DMUS_IO_BAND_ITEM_HEADER2 {
    MUSIC_TIME lBandTimeLogical;
    MUSIC_TIME lBandTimePhysical;
} DMUS_IO_BAND_ITEM_HEADER2;
```

Members

lBandTimeLogical
Time in the music with which the band change is associated.

lBandTimePhysical
Precise time when band change will take effect. Should be close to logical time.

Requirements

Header: Declared in dmusicf.h.

See Also

Segment Timing

DMUS_IO_BAND_TRACK_HEADER

Contains information about the default behavior of a band track. Used in the Band Track Form.

```
typedef struct _DMUS_IO_BAND_TRACK_HEADER {
    BOOL bAutoDownload;
} DMUS_IO_BAND_TRACK_HEADER;
```

Members

bAutoDownload

Flag for automatic downloading of instruments when a segment is played.

Remarks

For more information on automatic downloading, see Using Bands.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_BUFFER_ATTRIBUTES_HEADER

Describes attributes of a DirectSound buffer. Used in the Audiopath Form.

```
typedef struct _DMUS_IO_BUFFER_ATTRIBUTES_HEADER {
    GUID    guidBufferID;
    DWORD   dwFlags;
} DMUS_IO_BUFFER_ATTRIBUTES_HEADER;
```

Members

guidBufferID

Unique identifier of the buffer configuration. The following values are defined for standard buffer types.

GUID_Buffer_Reverb
GUID_Buffer_EnvReverb
GUID_Buffer_Stereo
GUID_Buffer_3D_Dry
GUID_Buffer_Mono

dwFlags

Flags describing the buffer. The following values are defined.

DMUS_BUFFERF_DEFINED
One of the standard buffer types.

DMUS_BUFFERF_MIXIN
Mix-in buffer.

DMUS_BUFFERF_SHARED
Buffer shared among audiopaths.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_CHORD

Contains information about a chord change. Used in the Chord Track List.

```
typedef struct _DMUS_IO_CHORD {
    WCHAR       wszName[16];
    MUSIC_TIME  mtTime;
    WORD        wMeasure;
    BYTE        bBeat;
    BYTE        bFlags
} DMUS_IO_CHORD;
```

Members

wszName
Name of the chord.

mtTime
Time of the chord.

wMeasure
Measure that the chord falls on.

bBeat
Beat that the chord falls on.

bFlags
Flags. The following value is defined.

DMUS_CHORDKEYF_SILENT
The chord is silent. See the Remarks for **DMUS_CHORD_KEY**.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_CHORDENTRY

Contains information about a chord entry. Used in the Chord Map Form.

```
typedef struct _DMUS_IO_CHORDENTRY {
    DWORD   dwFlags;
    WORD    wConnectionID;
} DMUS_IO_CHORDENTRY;
```

Members

dwFlags
Flag indicating whether the chord is a starting chord (bit 2 set) or an ending chord (bit 3 set) in the chord graph.

wConnectionID
Replaces the run-time pointer to *this*. Each chord entry is tagged with a unique connection identifier.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_CHORDMAP

Contains information about a chordmap. Used in the Chord Map Form.

```
typedef struct _DMUS_IO_CHORDMAP {
    WCHAR   wszLoadName[20];
    DWORD   dwScalePattern;
    DWORD   dwFlags;
} DMUS_IO_CHORDMAP;
```

Members

wszLoadName
Name of the chordmap, used in the object description when the chordmap is loaded.

dwScalePattern
Scale associated with the chordmap. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the scale.

dwFlags
Flags. Can be zero or the following value:

DMUS_CHORDMAPF_VERSION8
The chordmap was created for DirectX 8.0 or later.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_CHORDMAP_SIGNPOST

Contains information about a signpost chord in a chordmap. Used in the Chord Map Form.

```
typedef struct _DMUS_IO_CHORDMAP_SIGNPOST {
    DWORD   dwChords;
    DWORD   dwFlags;
} DMUS_IO_CHORDMAP_SIGNPOST;
```

Members

dwChords

Types of signpost supported by this chord. The values are used to match against the same values as they appear in templates. Composing from template consists of (among other things) looking for these values in the template and finding actual chords in the chordmap that match these values. The following flags are defined:

DMUS_SIGNPOSTF_A
DMUS_SIGNPOSTF_B
DMUS_SIGNPOSTF_C
DMUS_SIGNPOSTF_D
DMUS_SIGNPOSTF_E
DMUS_SIGNPOSTF_F
DMUS_SIGNPOSTF_LETTER
DMUS_SIGNPOSTF_1
DMUS_SIGNPOSTF_2
DMUS_SIGNPOSTF_3
DMUS_SIGNPOSTF_4
DMUS_SIGNPOSTF_5
DMUS_SIGNPOSTF_6
DMUS_SIGNPOSTF_7
DMUS_SIGNPOSTF_ROOT
DMUS_SIGNPOSTF_CADENCE

dwFlags

Flags defining whether this chord is to be preceded by cadence chords. Signpost chords can have up to two cadence chords. This value can be SPOST_CADENCE1 (first cadence), SPOST_CADENCE2 (second cadence), or a combination of these two flags.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_IO_SIGNPOST

DMUS_IO_CHORDMAP_SUBCHORD

Contains information about a subchord. Used in the Chord Map Form.

```
typedef struct _DMUS_IO_CHORDMAP_SUBCHORD {
    DWORD   dwChordPattern;
```

```
    DWORD   dwScalePattern;
    DWORD   dwInvertPattern;
    BYTE    bChordRoot;
    BYTE    bScaleRoot;
    WORD    wCFlags;
    DWORD   dwLevels;
} DMUS_IO_CHORDMAP_SUBCHORD;
```

Members

dwChordPattern
Notes in the subchord. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the chord.

dwScalePattern
Notes in the scale. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the scale.

dwInvertPattern
Points in the scale at which inversions can occur. Bits that are off signify that the notes in the interval cannot be inverted. Thus, the pattern 100001111111 indicates that inversions are allowed anywhere except between the fifth and seventh degrees of a major scale.

bChordRoot
Root of the subchord, where 0 is the lowest C in the range and 23 is the top B.

bScaleRoot
Root of the scale, where 0 is the lowest C in the range and 23 is the top B.

wCFlags
Reserved for future use.

dwLevels
Bit field showing which levels are supported by this subchord. Each part in a style is assigned a level, and this chord is used only for parts whose levels are contained in this member.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_SUBCHORD

DMUS_IO_COMMAND

Contains information about a command event. Used in the Command Track Chunk.

```
typedef struct _DMUS_IO_COMMAND {
    MUSIC_TIME  mtTime;
    WORD        wMeasure;
    BYTE        bBeat;
    BYTE        bCommand;
    BYTE        bGrooveLevel;
    BYTE        bGrooveRange;
    BYTE        bRepeatMode;
} DMUS_IO_COMMAND;
```

Members

mtTime
Time of the command.

wMeasure
Measure that the command falls on.

bBeat
Beat that the command falls on.

bCommand
Command type. See **DMUS_COMMANDT_TYPES**.

bGrooveLevel
Groove level, or 0 if the command is not a groove command.

bGrooveRange
Size of the range within which the groove level can be randomized. If this value is an odd number, the groove range is **bGrooveRange** – 1. For instance, if the groove level is 35 and **bGrooveRange** is 5, the adjusted groove range is 4 and the groove level could be anywhere from 33 to 37.

bRepeatMode
Flag that specifies how patterns are selected for repetition. See **DMUS_PATTERNT_TYPES**.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_CONTAINED_OBJECT_HEADER

Used before each object in a Container Form.

```
typedef struct _DMUS_IO_CONTAINED_OBJECT_HEADER {
    GUID        guidClassID;
    DWORD       dwFlags;
    FOURCC      ckid;
```

```
    FOURCC      fccType;
} DMUS_IO_CONTAINED_OBJECT_HEADER;
```

Members

guidClassID
Class identifier of the object.

dwFlags
Can be zero or the following flag.

DMUS_CONTAINED_OBJF_KEEP
Keep the object cached in the loader after the container is released.

ckid
Identifier of the data chunk. If this value is zero, it is assumed that the chunk is of type LIST, so **fccType** is valid and must be nonzero.

fccType
List type. If this value is zero, **ckid** is valid and must be nonzero.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_CONTAINER_HEADER

Used in the Container Form.

```
typedef struct _DMUS_IO_CONTAINER_HEADER {
    DWORD       dwFlags;
} DMUS_IO_CONTAINER_HEADER;
```

Members

dwFlags
DWORD value that specifies flags. Can be zero or the following value.

DMUS_CONTAINER_NOLOADS
Contained items are not loaded when the container is loaded. Entries are created in the loader, but the objects are not created until they are specifically loaded.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_CURVE_ITEM

Contains information about a curve event in a track. Used in the Sequence Track Chunk.

```
typedef struct _DMUS_IO_CURVE_ITEM {
    MUSIC_TIME  mtStart;
    MUSIC_TIME  mtDuration;
    MUSIC_TIME  mtResetDuration;
    DWORD       dwPChannel;
    short       nOffset;
    short       nStartValue;
    short       nEndValue;
    short       nResetValue;
    BYTE        bType;
    BYTE        bCurveShape;
    BYTE        bCCData;
    BYTE        bFlags;
    WORD        wParamType;
    WORD        wMergeIndex
} DMUS_IO_CURVE_ITEM;
```

Members

mtStart
Start time of the curve.

mtDuration
Duration of the curve.

mtResetDuration
Time after the curve is finished during which a reset can occur.

dwPChannel
Performance channel for the event.

nOffset
Offset from the grid boundary at which the curve occurs, in music time. Because MIDI curves are associated with the closest grid when loaded, this value can be positive or negative.

nStartValue
Start value.

nEndValue
End value.

nResetValue
Reset value, set upon a flush or invalidation within the time set by **mtResetDuration**.

bType

Type of curve. The following types are defined:

DMUS_CURVET_CCCURVE
Continuous controller curve (MIDI Control Change channel voice message; status byte &HB*n*, where *n* is the channel number).

DMUS_CURVET_MATCURVE
Monophonic aftertouch curve (MIDI Channel Pressure channel voice message; status byte &HD*n*).

DMUS_CURVET_PATCURVE
Polyphonic aftertouch curve (MIDI Poly Key Pressure channel voice message, status byte &HD*n*).

DMUS_CURVET_PBCURVE
Pitch-bend curve (MIDI Pitch Bend channel voice message; status byte &HE*n*).

DMUS_CURVET_NRPNCURVE
NRPN curve.

DMUS_CURVET_RPNCURVE
RPN curve.

bCurveShape
Shape of curve. The following shapes are defined:

DMUS_CURVES_EXP
Exponential curve shape.

DMUS_CURVES_INSTANT
Instant curve shape (beginning and end of curve happen at essentially the same time).

DMUS_CURVES_LINEAR
Linear curve shape.

DMUS_CURVES_LOG
Logarithmic curve shape.

DMUS_CURVES_SINE
Sine curve shape.

bCCData
CC number if this is a control change type.

bFlags
Set to DMUS_CURVE_RESET if the **nResetValue** must be set when an invalidation occurs because of a transition. If 0, the curve stays permanently at the new value. All other bits are reserved.

wParamType
Parameter number for RPN and NRPN types.

wMergeIndex
Merge index. Supported for mod wheel, reverb send, chorus send, pitch bend, volume, and expression controllers.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_IO_SEQ_ITEM, Curves

DMUS_IO_INSTRUMENT

Contains information about an instrument. Used in the Band Form.

```
typedef struct _DMUS_IO_INSTRUMENT {
    DWORD  dwPatch;
    DWORD  dwAssignPatch;
    DWORD  dwNoteRanges[4];
    DWORD  dwPChannel;
    DWORD  dwFlags;
    BYTE   bPan;
    BYTE   bVolume;
    short  nTranspose;
    DWORD  dwChannelPriority;
    short  nPitchBendRange;
} DMUS_IO_INSTRUMENT;
```

Members

dwPatch
MSB, LSB, and program change to define instrument.

dwAssignPatch
MSB, LSB, and program change to assign to instrument when downloading.

dwNoteRanges
128 bits; one for each MIDI note that the instrument must be able to play.

dwPChannel
Performance channel that the instrument plays on.

dwFlags
Control flags. The following values are defined:

DMUS_IO_INST_ASSIGN_PATCH
The **dwAssignPatch** member is valid.

DMUS_IO_INST_BANKSELECT
The **dwPatch** member contains a valid bank select, both MSB and LSB.

DMUS_IO_INST_CHANNEL_PRIORITY
The **dwChannelPriority** member is valid.

DMUS_IO_INST_GM
Instrument is from the General MIDI collection.

DMUS_IO_INST_GS
Instrument is from the Roland GS collection.

DMUS_IO_INST_NOTERANGES
The **dwNoteRanges** member is valid.

DMUS_IO_INST_PAN
The **bPan** member is valid.

DMUS_IO_INST_PATCH
The **dwPatch** member is valid.

DMUS_IO_INST_PITCHBENDRANGE
The **nPitchBendRange** member is valid.

DMUS_IO_INST_TRANSPOSE
The **nTranspose** member is valid.

DMUS_IO_INST_USE_DEFAULT_GM_SET
The default General MIDI instrument set should be downloaded to the port, even if the port has GM in hardware. When a MIDI file that contains an XG or GS reset is parsed, the bank-select message is sent, whether or not GUID_StandardMIDIFile was commanded on the band. In other words, GUID_StandardMIDIFile is effective only for pure GM files.

DMUS_IO_INST_VOLUME
The **bVolume** member is valid.

DMUS_IO_INST_XG
Instrument is from the Yamaha XG collection.

bPan
Pan for the instrument.

bVolume
Volume for the instrument.

nTranspose
Number of semitones to transpose notes.

dwChannelPriority
Channel priority. For a list of defined values, see **IDirectMusicPort8::GetChannelPriority**.

nPitchBendRange
Number of semitones shifted by pitch bend.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_LYRICSTRACK_EVENT HEADER

Used in a Lyrics Track List.

```
typedef struct _DMUS_IO_LYRICSTRACK_EVENTHEADER {
    DWORD dwFlags;
    DWORD dwTimingFlags;
    MUSIC_TIME lTimeLogical;
    MUSIC_TIME lTimePhysical;
} DMUS_IO_LYRICSTRACK_EVENTHEADER;
```

Members

dwFlags
Reserved; must be zero.

dwTimingFlags
Flags to determine the timing of the notification. Can be one or more of the following members of the **DMUS_PMSGF_FLAGS** enumeration.

DMUS_PMSGF_TOOL_IMMEDIATE
Message should be processed immediately, regardless of its time stamp.

DMUS_PMSGF_TOOL_QUEUE
Message should be processed just before its time stamp, allowing for port latency.

DMUS_PMSGF_TOOL_ATTIME
Message should be processed at the time stamp.

lTimeLogical
Time in the music with which the event is associated.

lTimePhysical
Precise time when the event will be triggered. This should be close to logical time.

Requirements

Header: Declared in dmusicf.h.

See Also

Logical Time vs. Actual Time

DMUS_IO_MELFORM

Not implemented in DirectX 8.0.

Used in the Melody Formulation Track List.

```
typedef struct _DMUS_IO_MELFORM {
    DWORD  dwPlaymode;
} DMUS_IO_MELFORM;
```

Members

dwPlaymode
Reserved. Must be 0.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_MOTIFSETTINGS

Contains information about a motif. Used in the Style Form.

```
typedef struct _DMUS_IO_MOTIFSETTINGS {
    DWORD       dwRepeats;
    MUSIC_TIME  mtPlayStart;
    MUSIC_TIME  mtLoopStart;
    MUSIC_TIME  mtLoopEnd;
    DWORD       dwResolution;
} DMUS_IO_MOTIFSETTINGS;
```

Members

dwRepeats
Number of repetitions.

mtPlayStart
Start of playback, normally 0.

mtLoopStart
Start of looping portion, normally 0.

mtLoopEnd
End of looping portion. Must be greater than **mtLoopStart,** or zero to loop the entire motif.

dwResolution
Default resolution. See **DMUS_TIME_RESOLVE_FLAGS**.

Requirements

Header: Declared in dmusicf.h.

See Also

IDirectMusicSegment8::SetLoopPoints

DMUS_IO_MUTE

Contains information about a mute event on a channel. Used in the Mute Track Chunk.

```
typedef struct _DMUS_IO_MUTE {
    MUSIC_TIME  mtTime;
    DWORD       dwPChannel;
    DWORD       dwPChannelMap;
} DMUS_IO_MUTE;
```

Members

mtTime
Time of the event.

dwPChannel
Performance channel to mute or remap.

dwPChannelMap
Channel to which **dwPChannel** is being mapped, or 0xFFFFFFFF if **dwPChannel** is to be muted.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_MUTE_PARAM

DMUS_IO_NEXTCHORD

Contains information about the next chord in a chord graph. Used in the Chord Map Form.

```
typedef struct _DMUS_IO_NEXTCHORD {
    DWORD   dwFlags;
    WORD    nWeight;
    WORD    wMinBeats;
    WORD    wMaxBeats;
    WORD    wConnectionID;
} DMUS_IO_NEXTCHORD;
```

Members

dwFlags

Reserved for future use.

nWeight
Likelihood (in the range from 1 through 100) that this link is followed when traversing the chord graph.

wMinBeats
Smallest number of beats that this chord is allowed to play in a composed segment.

wMaxBeats
Largest number of beats that this chord is allowed to play in a composed segment.

wConnectionID
Refers to the **wConnectionID** member of a **DMUS_IO_CHORDENTRY** structure.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_PARAMCONTROLTRACK_CURVEINFO

Used in a Parameter Control Track List.

```
typedef struct _DMUS_IO_PARAMCONTROLTRACK_CURVEINFO {
    MUSIC_TIME  mtStartTime;
    MUSIC_TIME  mtEndTime;
    float       fltStartValue;
    float       fltEndValue;
    DWORD       dwCurveType;
    DWORD       dwFlags;
} DMUS_IO_PARAMCONTROLTRACK_CURVEINFO;
```

Members

mtStartTime
Start time of the curve.

mtEndTime
End time of the curve.

fltStartValue
Start value of the curve.

fltEndValue
End value of the curve.

dwCurveType

Item from the **MP_CURVE_TYPE** enumeration. See **MP_CURVE_TYPE** in the DirectShow documentation.

dwFlags
Combination of the MPF_ENVLP_* constants. See **Envelope Flags** in the DirectShow documentation.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_PARAMCONTROLTRACK_OBJECTHEADER

Used in the Parameter Control Track List.

```
typedef struct _DMUS_IO_PARAMCONTROLTRACK_OBJECTHEADER {
    DWORD dwFlags;
    GUID guidTimeFormat;
    DWORD dwPChannel;
    DWORD dwStage;
    DWORD dwBuffer;
    GUID guidObject;
    DWORD dwIndex;
} DMUS_IO_PARAMCONTROLTRACK_OBJECTHEADER;
```

Members

dwFlags
Reserved; must be zero.

guidTimeFormat
Time format to set the object to. Must be GUID_TIME_REFERENCE or GUID_TIME_MUSIC, which are defined in Medparam.h.

dwPChannel
Performance channel, 0, or DMUS_PCHANNEL_ALL.

dwStage
Stage in the path.

dwBuffer
Index of the buffer, if there is more than one.

guidObject
Class identifier of the object, such as GUID_DSFX_STANDARD_CHORUS.

dwIndex
Index of the object in the list of matching objects.

Remarks

For more information on the possible values for each member, see **IDirectMusicSegmentState8::GetObjectInPath**.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_PARAMCONTROLTRACK_PARAMHEADER

Used in the Parameter Control Track List.

```
typedef struct _DMUS_IO_PARAMCONTROLTRACK_PARAMHEADER {
    DWORD dwFlags;
    DWORD dwIndex;
} DMUS_IO_PARAMCONTROLTRACK_PARAMHEADER;
```

Members

dwFlags
Reserved; must be zero.

dwIndex
Index number of the parameter on the object.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_PARTREF

Contains information about a part reference. Used in the Style Form.

```
typedef struct _DMUS_IO_PARTREF {
    GUID  guidPartID;
    WORD  wLogicalPartID;
    BYTE  bVariationLockID;
    BYTE  bSubChordLevel;
    BYTE  bPriority;
    BYTE  bRandomVariation;
    WORD  wPad;
    DWORD dwPChannel;
} DMUS_IO_PARTREF;
```

Members

guidPartID
Identifier of the part.

wLogicalPartID
Identifier corresponding to a particular MIDI channel on a port. This member has been superseded by **dwPChannel** and is no longer used.

bVariationLockID
Variation lock identifier. Parts with the same value in this member always play the same variation. A value of 0 means that the part plays its variations independently of all other parts.

bSubChordLevel
Subchord level that this part wants. See Remarks.

bPriority
Reserved for future use.

bRandomVariation
Can be 0, meaning that matching variations play sequentially, or one of the members of the **DMUS_VARIATIONT_TYPES** enumeration.

wPad
Padding for alignment; value not used.

dwPChannel
Performance channel of the part.

Remarks

The **bSubChordLevel** member contains a zero-based index value. At run time, 1 is shifted left by this value to yield a 1-bit value for comparison with the **dwLevels** member of a **DMUS_SUBCHORD** structure. Thus, a part with a **bSubChordLevel** of 0 would be mapped to any subchord that contained 1 in **dwLevels**.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_PATTERN

Contains information about a pattern. Used in the Style Form.

```
typedef struct _DMUS_IO_PATTERN {
    DMUS_IO_TIMESIG  timeSig;
    BYTE             bGrooveBottom;
    BYTE             bGrooveTop;
    WORD              wEmbellishment;
    WORD              wNbrMeasures;
    BYTE             bDestGrooveBottom;
```

```
    BYTE            bDestGrooveTop;
    DWORD           dwFlags;
} DMUS_IO_PATTERN;
```

Members

timeSig
DMUS_IO_TIMESIG structure containing a time signature to override the style's default time signature.

bGrooveBottom
Bottom of the groove range.

bGrooveTop
Top of the groove range.

wEmbellishment
Type of embellishment. One of the constants from the **DMUS_COMMANDT_TYPES** enumeration, or a value defined by the content provider.

wNbrMeasures
Length of the pattern in measures.

bDestGrooveBottom
Bottom of groove range for next pattern.

bDestGrooveTop
Top of groove range for next pattern.

dwFlags
Flags. Can be zero or the following value:

DMUS_PATTERNF_PERSIST_CONTROL
Variation settings in the state data of a pattern-based track persist in the track after it stops playing.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_PCHANNELTOBUFFER_HEADER

Defines a range of performance channels and the buffers they connect to. Used in the port configuration list of an Audiopath Form.

```
typedef struct _DMUS_IO_PCHANNELTOBUFFER_HEADER {
    DWORD  dwPChannelBase;
    DWORD  dwPChannelCount;
    DWORD  dwBufferCount;
    DWORD  dwFlags;
```

```
} DMUS_IO_PCHANNELTOBUFFER_HEADER;
```

Members

dwPChannelBase
First performance channel.

dwPChannelCount
Number of performance channels.

dwBufferCount
Number of buffers the channels connect to.

dwFlags
Reserved. Must be 0.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_PLAYMARKER

Contains information about a play marker, which is a point within a primary segment where a new segment is permitted to start playing. Used in the Marker Track List.

```
typedef struct _DMUS_IO_PLAY_MARKER {
    MUSIC_TIME mtTime;
} DMUS_IO_PLAY_MARKER;
```

Members

mtTime
Time of legal play point.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_PORTCONFIG_HEADER

Used in the Audiopath Form.

```
typedef struct _DMUS_IO_PORTCONFIG_HEADER {
    GUID    guidPort;
    DWORD   dwPChannelBase;
    DWORD   dwPChannelCount;
    DWORD   dwFlags;
} DMUS_IO_PORTCONFIG_HEADER;
```

Members

guidPort
Unique identifier of port.

dwPChannelBase
First performance channel.

dwPChannelCount
Number of performance channels.

dwFlags
Configuration flags. The following values are defined.

DMUS_PORTCONFIGF_DRUMSON10
Drums are on channel 10.

DMUS_PORTCONFIGF_USEDEFAULT
Use the default port.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_REFERENCE

Contains information about a reference to another object that might be stored in another file. Used in the Reference List chunk.

```
typedef struct _DMUS_IO_REFERENCE {
    GUID    guidClassID;
    DWORD   dwValidData;
} DMUS_IO_REFERENCE;
```

Members

guidClassID
Class identifier.

dwValidData
Flags to indicate which data chunks for the reference are present. For a list of values, see the corresponding member of **DMUS_OBJECTDESC**.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_SCRIPT_HEADER

Used in the Script Form.

```
typedef struct _DMUS_IO_SCRIPT_HEADER {
    DWORD  dwFlags;
} DMUS_IO_SCRIPT_HEADER;
```

Members

dwFlags

DWORD value that specifies the loading behavior of the script. Can be one or more of the following values.

DMUS_SCRIPTIOF_LOAD_ALL_CONTENT
All content in the script's container is loaded.

DMUS_SCRIPTIOF_DOWNLOAD_ALL_SEGMENTS
If DMUS_SCRIPTIOF_LOAD_ALL_CONTENT is set, the bands from all the segments in the script's container are downloaded when the script is initialized. Otherwise, a segment's bands are downloaded when the script loads the segment.

If DMUS_SCRIPTIOF_DOWNLOAD_ALL_SEGMENTS is not set, the script must manually download and unload the segment's bands.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_SCRIPTTRACK_EVENT HEADER

Used in a Script Track List.

```
typedef struct _DMUS_IO_SCRIPTTRACK_EVENTHEADER {
    DWORD dwFlags;
    MUSIC_TIME lTimeLogical;
    MUSIC_TIME lTimePhysical;
} DMUS_IO_SCRIPTTRACK_EVENTHEADER;
```

Members

dwFlags

Flag that determines when the event is set. Can be one of the following values.

DMUS_IO_SCRIPTTRACKF_PREPARE

Fire the event in advance of the time stamp, at prepare time. This is the default because it leaves the script enough time to change the music happening at the target time.

DMUS_IO_SCRIPTTRACKF_QUEUE
Fire the event just before the time stamp, at queue time.

DMUS_IO_SCRIPTTRACKF_ATTIME
Fire the event exactly at the time stamp.

lTimeLogical
Logical time of the event.

lTimePhysical
Actual time of the event.

Requirements

Header: Declared in dmusicf.h.

See Also

Logical Time vs. Actual Time

DMUS_IO_SEGMENT_HEADER

Contains information about a segment. Used in the Segment Form.

```
typedef struct _DMUS_IO_SEGMENT_HEADER {
    DWORD        dwRepeats;
    MUSIC_TIME   mtLength;
    MUSIC_TIME   mtPlayStart;
    MUSIC_TIME   mtLoopStart;
    MUSIC_TIME   mtLoopEnd;
    DWORD        dwResolution;
    REFERENCE_TIME rtLength;
    DWORD        dwFlags;
    DWORD        dwReserved;
} DMUS_IO_SEGMENT_HEADER;
```

Members

dwRepeats
Number of repetitions.

mtLength
Length of the segment.

mtPlayStart
Start of playback, normally 0.

mtLoopStart
Start of the looping portion, normally 0.

mtLoopEnd
End of the looping portion. Must be greater than **mtPlayStart,** or zero to loop the entire segment.

dwResolution
Default resolution. See **DMUS_TIME_RESOLVE_FLAGS**.

rtLength
Length of the segment in reference time. Valid if the DMUS_SEGIOF_REFLENGTH flag is set.

dwFlags
Can be zero or the following flag.

DMUS_SEGIOF_REFLENGTH
The value in **rtLength** overrides **mtLength**.

dwReserved
Reserved.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_IO_MOTIFSETTINGS, **IDirectMusicSegment8::SetLoopPoints**.

DMUS_IO_SEGMENT_ITEM_HEADER

Contains information about a segment referenced in the Segment Trigger Track List.

```
typedef struct _DMUS_IO_SEGMENT_ITEM_HEADER{
    MUSIC_TIME      lTimeLogical;
    MUSIC_TIME      lTimePhysical;
    DWORD           dwPlayFlags;
    DWORD           dwFlags;
} DMUS_IO_SEGMENT_ITEM_HEADER;
```

Members

lTimeLogical
Time in the music with which the event is associated.

lTimePhysical
Actual time at which the segment is to play.

dwPlayFlags
Flags that will be passed to **IDirectMusicPerformance8::PlaySegmentEx**. See **DMUS_SEGF_FLAGS**.

dwFlags
Can be zero or the following value.

DMUS_SEGMENTTRACKF_MOTIF
The DMRF chunk is a link to a style, and the 'snam' chunk is the name of a motif within the style.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_SEGMENT_TRACK_HEADER

Contains information about a Segment Trigger Track List.

```
typedef struct _DMUS_IO_SEGMENT_TRACK_HEADER {
    DWORD dwFlags;
} DMUS_IO_SEGMENT_TRACK_HEADER;
```

Members

dwFlags
Reserved. Must be zero.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_SEGREF_HEADER

Not implemented in DirectX 8.0.

Contains information about a segment reference in a Song Form.

```
typedef struct _DMUS_IO_SEGREF_HEADER {
    DWORD       dwID;
    DWORD       dwSegmentID;
    DWORD       dwToolGraphID;
    DWORD       dwFlags;
    DWORD       dwNextPlayID;
} DMUS_IO_SEGREF_HEADER;
```

Members

dwID
Unique identifier of the segment. Must not be greater than DMUS_SONG_MAXSEGID.

dwSegmentID
Identifier of an optional source segment to link to.

dwToolGraphID
Identifier of an optional toolgraph to use for processing.

dwFlags
Reserved. Must be zero.

dwNextPlayID
Identifier of next segment in song chain.

DMUS_IO_SEQ_ITEM

Contains information about an item of data in a sequence track. Used in the Sequence Track Chunk.

```
typedef struct _DMUS_IO_SEQ_ITEM {
    MUSIC_TIME  mtTime;
    MUSIC_TIME  mtDuration;
    DWORD       dwPChannel;
    short       nOffset
    BYTE        bStatus;
    BYTE        bByte1;
    BYTE        bByte2;
} DMUS_IO_SEQ_ITEM;
```

Members

mtTime
Logical time of the event.

mtDuration
Duration for which the event is valid.

dwPChannel
Performance channel for the event.

nOffset
Offset from **mtTime** at which the note is played, in music time.

bStatus
MIDI event type. Equivalent to the MIDI status byte, but without channel information.

bByte1
First byte of the MIDI data.

bByte2
Second byte of the MIDI data.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_IO_CURVE_ITEM, MIDI Messages

DMUS_IO_SIGNPOST

Contains information about a signpost in a signpost track to associate it with signpost chords in a chordmap. Used in the Signpost Track Chunk.

```
typedef struct _DMUS_IO_SIGNPOST {
    MUSIC_TIME  mtTime;
    DWORD       dwChords;
    WORD        wMeasure;
} DMUS_IO_SIGNPOST;
```

Members

mtTime
Time of the signpost.

dwChords
Types of signpost chords allowed to be associated with this signpost. The values are used to match against the same values as they appear in templates. Composing from a template consists of (among other things) looking for these values in the template and finding actual chords in the chordmap that match these values. The following flags are defined:

DMUS_SIGNPOSTF_A
DMUS_SIGNPOSTF_B
DMUS_SIGNPOSTF_C
DMUS_SIGNPOSTF_D
DMUS_SIGNPOSTF_E
DMUS_SIGNPOSTF_F
DMUS_SIGNPOSTF_LETTER
DMUS_SIGNPOSTF_1
DMUS_SIGNPOSTF_2
DMUS_SIGNPOSTF_3
DMUS_SIGNPOSTF_4

DMUS_SIGNPOSTF_5
DMUS_SIGNPOSTF_6
DMUS_SIGNPOSTF_7
DMUS_SIGNPOSTF_ROOT
DMUS_SIGNPOSTF_CADENCE

wMeasure
Measure on which the signpost falls.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_IO_CHORDMAP_SIGNPOST

DMUS_IO_SONG_HEADER

Not implemented in DirectX 8.0.

Contains information about a Song Form.

```
typedef struct _DMUS_IO_SONG_HEADER {
    DWORD  dwFlags;
    DWORD  dwStartSegID;
} DMUS_IO_SONG_HEADER;
```

Members

dwFlags
Reserved.

dwStartSegID
Identifier of the segment to play first.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_STYLE

Contains information about the time signature and tempo of a style. Used in the Style Form.

```
typedef struct _DMUS_IO_STYLE {
    DMUS_IO_TIMESIG  timeSig;
```

```
    double          dblTempo;
} DMUS_IO_STYLE;
```

Members

timeSig
DMUS_IO_TIMESIG structure containing the default time signature for the style.

dblTempo
Tempo of the style.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_STYLE_ANTICIPATION

Describes a resolution anticipation. Used in the Style Form.

```
typedef struct _DMUS_IO_STYLE_ANTICIPATION {
    MUSIC_TIME    mtGridStart;
    DWORD         dwVariation;
    short         nTimeOffset;
    BYTE          bTimeRange;
} DMUS_IO_STYLE_ANTICIPATION;
```

Members

mtGridStart
Offset within the part, in grids, at which the event is to play. See Remarks.

dwVariation
Variations, where each bit set specifies a valid variation.

nTimeOffset
Offset of the time from **mtGridStart**.

bTimeRange
Range by which to randomize time. See Remarks.

Remarks

The time of the event can be calculated as follows, where **TimeSig** is a **DMUS_IO_TIMESIG** structure containing the time signature.

```
mtEventTime = nTimeOffset + ((mtGridStart / TimeSig.wGridsPerBeat) *
((DMUS_PPQ * 4) / TimeSig.bBeat) + (mtGridStart %
TimeSig.wGridsPerBeat) * (((DMUS_PPQ * 4)/ TimeSig.bBeat) /
```

```
TimeSig.wGridsPerBeat))
```

The value in **bTimeRange** is converted to music time when the event occurs, according to the formula given in the Remarks to **DMUS_IO_STYLENOTE**.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_STYLECURVE

Contains information about a curve in a style. Used in the Style Form.

```
typedef struct _DMUS_IO_STYLECURVE {
    MUSIC_TIME  mtGridStart;
    DWORD       dwVariation;
    MUSIC_TIME  mtDuration;
    MUSIC_TIME  mtResetDuration;
    short       nTimeOffset;
    short       nStartValue;
    short       nEndValue;
    short       nResetValue;
    BYTE        bEventType;
    BYTE        bCurveShape;
    BYTE        bCCData;
    BYTE        bFlags;
    WORD        wParamType;
    WORD        wMergeIndex;
} DMUS_IO_STYLECURVE;
```

Members

mtGridStart
Offset, in grids, at which the curve occurs.

dwVariation
Variations that this curve belongs to. Each bit corresponds to one of 32 variations.

mtDuration
Duration of the curve.

mtResetDuration
Time after the curve is finished during which a reset can occur.

nTimeOffset
Offset from **mtGridStart** at which the curve occurs. See the Remarks for **DMUS_IO_STYLE_ANTICIPATION**.

nStartValue
Start value.

nEndValue
End value.

nResetValue
Reset value, set upon a flush or invalidation during the time specified by **mtResetDuration**.

bEventType
Type of curve. See **DMUS_IO_CURVE_ITEM**.

bCurveShape
Shape of curve. See **DMUS_IO_CURVE_ITEM**.

bCCData
CC number if this is a control change type.

bFlags
Set to DMUS_CURVE_RESET if the **nResetValue** must be set when an invalidation occurs because of a transition. If 0, the curve stays permanently at the new value. All other bits are reserved.

wParamType
RPN or NRPN parameter number.

wMergeIndex
Merge index. Supported for mod wheel, reverb send, chorus send, pitch bend, volume, and expression controllers.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_CURVE_PMSG, **DMUS_IO_CURVE_ITEM**

DMUS_IO_STYLEMARKER

Contains information about a marker in a style. Used in the Style Form.

```
typedef struct _DMUS_IO_STYLEMARKER {
    MUSIC_TIME  mtGridStart;
    DWORD       dwVariation;
    WORD        wMarkerFlags;
} DMUS_IO_STYLEMARKER;
```

Members

mtGridStart
Offset, in grids, at which the marker occurs.

dwVariation

Variations that this marker belongs to. Each bit corresponds to one of 32 variations.

wMarkerFlags

Flags that specify behavior of the marker. Can be zero or one or more of the following values. If zero, the behavior is as it was in DirectX version 7.0.

DMUS_MARKERF_START
Start a variation.

DMUS_MARKERF_STOP
Stop a variation.

DMUS_MARKERF_CHORD_ALIGN
New variations must align with a chord. This flag is ignored unless combined with one or both of DMUS_MARKERF_START and DMUS_MARKERF_STOP.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_STYLENOTE

Contains information about a note in a style. Used in the Style Form.

```
typedef struct _DMUS_IO_STYLENOTE {
    MUSIC_TIME  mtGridStart;
    DWORD       dwVariation;
    MUSIC_TIME  mtDuration;
    short       nTimeOffset;
    WORD        wMusicValue;
    BYTE        bVelocity;
    BYTE        bTimeRange;
    BYTE        bDurRange;
    BYTE        bVelRange;
    BYTE        bInversionID;
    BYTE        bPlayModeFlags;
    BYTE        bNoteFlags;
} DMUS_IO_STYLENOTE;
```

Members

mtGridStart

Offset, in grids, at which the note occurs.

dwVariation

Variations that this note belongs to. Each bit corresponds to one of 32 variations.

mtDuration

Duration of the note.

nTimeOffset
Time after **mtGridStart** at which the event occurs. See the Remarks for **DMUS_IO_STYLE_ANTICIPATION**.

wMusicValue
Position in the scale.

bVelocity
Note velocity.

bTimeRange
Range within which to randomize start time. See Remarks.

bDurRange
Range within which to randomize duration. See Remarks.

bVelRange;
Range within which to randomize velocity.

bInversionID
Identifier of inversion group to which this note belongs.

bPlayModeFlags
Flags to override the play mode of the part. For a list of values, see **DMUS_PLAYMODE_FLAGS**.

bNoteFlags
Flags. See **DMUS_NOTEF_FLAGS**.

Remarks

The values in **bTimeRange** and **bDurRange** are converted to music time when the note is played, using the following function:

```
int StoredRangeToActualRange(BYTE bRange)
{
    int nResult = 0;
    if (0 <= bRange && bRange <= 190)
    {
        nResult = bRange;
    }
    else if (191 <= bRange && bRange <= 212)
    {
        nResult = ((bRange - 190) * 5) + 190;
    }
    else if (213 <= bRange && bRange <= 232)
    {
        nResult = ((bRange - 212) * 10) + 300;
    }
    else // bRange > 232
    {
        nResult = ((bRange - 232) * 50) + 500;
    }
```

```
    return nResult;
}
```

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_STYLEPART

Contains information about a musical part. Used in the Style Form.

```
typedef struct _DMUS_IO_STYLEPART {
    DMUS_IO_TIMESIG  timeSig;
    DWORD            dwVariationChoices[32];
    GUID             guidPartID;
    WORD             wNbrMeasures;
    BYTE             bPlayModeFlags;
    BYTE             bInvertUpper;
    BYTE             bInvertLower;
    BYTE             bPad[3];
    DWORD            dwFlags;
} DMUS_IO_STYLEPART;
```

Members

timeSig

DMUS_IO_TIMESIG structure containing a time signature to override the style's default time signature.

dwVariationChoices

Each element corresponds to one of 32 possible variations. The flags set in each element indicate which types of chord are supported by that variation (see Remarks). One of the mode masks is also set to indicate whether the variations are in DirectMusic or IMA mode.

The following flags are defined:

DMUS_VARIATIONF_MAJOR
Seven positions in the scale for major chords.

DMUS_VARIATIONF_MINOR
Seven positions in the scale for minor chords.

DMUS_VARIATIONF_OTHER
Seven positions in the scale for other chords.

DMUS_VARIATIONF_ROOT_SCALE
Handles chord roots in the scale.

DMUS_VARIATIONF_ROOT_FLAT
Handles flat chord roots (based on scale notes).

DMUS_VARIATIONF_ROOT_SHARP

Handles sharp chord roots (based on scale notes).

DMUS_VARIATIONF_TYPE_TRIAD
Handles simple chords for triads.

DMUS_VARIATIONF_TYPE_6AND7
Handles simple chords for 6 and 7.

DMUS_VARIATIONF_TYPE_COMPLEX
Handles complex chords.

DMUS_VARIATIONF_DEST_TO1
Handles transitions to the 1 chord.

DMUS_VARIATIONF_DEST_TO5
Handles transitions to the 5 chord.

DMUS_VARIATIONF_MODES
Mode mask. Obsolete.

DMUS_VARIATIONF_MODES_EX
Mode mask.

DMUS_VARIATIONF_IMA25_MODE
Mode mask. If (**dwVariationChoices** & DMUS_VARIATIONF_MODES_EX) == DMUS_VARIATIONF_IMA25_MODE, the variations are in Interactive Music Architecture mode.

DMUS_VARIATIONF_DMUS_MODE
Mode mask. If **dwVariationChoices** contains this mask, the variations are in DirectMusic mode. All variations authored in DirectMusic Producer use this mode.

guidPartID
Unique identifier of the part.

wNbrMeasures
Length of the part, in measures.

bPlayModeFlags
Flags to define the play mode. For a list of values, see **DMUS_PLAYMODE_FLAGS**.

bInvertUpper
Upper limit of inversion.

bInvertLower
Lower limit of inversion.

bPad
Unused.

dwFlags
Flags that specify the behavior of the part. Can include the following values:

DMUS_PARTF_USE_MARKERS
Part uses marker events.

DMUS_PARTF_ALIGN_CHORDS
Part is allowed to switch only on chord-aligned markers.

Remarks

The flags in **dwVariationChoices** determine the types of chords supported by a given variation in DirectMusic mode. The first seven flags (bits 1 through 7) are set if the variation supports major chords rooted in scale positions. For example, if bits 1, 2, and 4 are set, the variation supports major chords rooted in the tonic, second, and fourth scale positions. The next seven flags serve the same purpose for minor chords, and the following seven flags serve the same purpose for chords that are not major or minor (for example, SUS 4 chords). Bits 22, 23, and 24 are set if the variation supports chords rooted in the scale, chords rooted sharp of scale tones, and chords rooted flat of scale tones, respectively. For example, to support a C# minor chord in the scale of C major, bits 8 (for tonic minor) and 24 (for sharp) must be set. Bits 25, 26, and 27 handle chords that are triads, sixth or seventh chords, and chords with extensions, respectively. Bits 28 and 29 handle chords that are followed by tonic and dominant chords, respectively.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_STYLERESOLUTION

Describes a style resolution. Used in the Style Form.

```
typedef struct _DMUS_IO_STYLERESOLUTION {
    DWORD  dwVariation;
    WORD   wMusicValue;
    BYTE   bInversionID;
    BYTE   bPlayModeFlags;
} DMUS_IO_STYLERESOLUTION;
```

Members

dwVariation
Variations, where each bit specifies a valid variation.

wMusicValue
Position in scale.

bInversionID
Inversion group to which this note belongs.

bPlayModeFlags
Play mode flags. See **DMUS_PLAYMODE_FLAGS**.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_SUBCHORD

Contains information about a subchord. Used in the Chord Track List.

```
typedef struct _DMUS_IO_SUBCHORD {
    DWORD   dwChordPattern;
    DWORD   dwScalePattern;
    DWORD   dwInversionPoints;
    DWORD   dwLevels;
    BYTE    bChordRoot;
    BYTE    bScaleRoot;
} DMUS_IO_SUBCHORD;
```

Members

dwChordPattern
Notes in the subchord. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the chord.

dwScalePattern
Notes in the scale. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the scale.

dwInversionPoints
Points in the scale at which inversions can occur. Bits that are off signify that the notes in the interval cannot be inverted. Thus, the pattern 100001111111 indicates that inversions are allowed anywhere except between the fifth and seventh degrees of a major scale.

dwLevels
Which levels are supported by this subchord. Certain instruments can be assigned different levels (such as to play only the lower subchords of a chord), and this value is a way of mapping subchords to those levels.

bChordRoot
Root of the subchord, where 0 is the lowest C in the range and 23 is the top B.

bScaleRoot
Root of the scale, where 0 is the lowest C in the range and 23 is the top B.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_SUBCHORD

DMUS_IO_SYSEX_ITEM

Contains information about a system exclusive MIDI message. Used in the Sysex Track Chunk.

```
typedef struct _DMUS_IO_SYSEX_ITEM {
    MUSIC_TIME  mtTime;
    DWORD       dwPChannel;
    DWORD       dwSysExLength;
} DMUS_IO_SYSEX_ITEM;
```

Members

mtTime
Time of the message.

dwPChannel
Performance channel of the event.

dwSysExLength
Length of the data, in bytes.

Requirements

Header: Declared in dmusicf.h.

See Also

MIDI System Messages

DMUS_IO_TEMPO_ITEM

Contains information about a tempo change in a track. Used in the Tempo Track Chunk.

```
typedef struct _DMUS_IO_TEMPO_ITEM {
    MUSIC_TIME  lTime;
    double      dblTempo;
} DMUS_IO_TEMPO_ITEM;
```

Members

lTime
Time of the tempo change.

dblTempo
Tempo, in beats per minute.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_TIMESIG

Contains information about the time signature of a segment. Used in the **DMUS_IO_STYLE**, **DMUS_IO_VERSION**, and **DMUS_IO_PATTERN** structures.

```
typedef struct _DMUS_IO_TIMESIG {
    BYTE  bBeatsPerMeasure;
    BYTE  bBeat;
    WORD  wGridsPerBeat;
} DMUS_IO_TIMESIG;
```

Members

bBeatsPerMeasure
Beats per measure (top of time signature).

bBeat
Note that receives the beat (bottom of the time signature), where 1 is a whole note, 2 is a half note, 4 is a quarter note, and so on. Zero is a 256th note.

wGridsPerBeat
Grids (subdivisions) per beat.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_IO_TIMESIGNATURE_ITEM

DMUS_IO_TIMESIGNATURE_ITEM

Contains information about a time signature change. Used in the Time Signature Track List.

```
typedef struct _DMUS_IO_TIMESIGNATURE_ITEM {
    MUSIC_TIME  lTime;
    BYTE        bBeatsPerMeasure;
    BYTE        bBeat;
    WORD        wGridsPerBeat;
} DMUS_IO_TIMESIGNATURE_ITEM;
```

Members

lTime
Time of the event.

bBeatsPerMeasure
Beats per measure (top of time signature).

bBeat
Note that receives the beat (bottom of the time signature), where 1 is a whole note, 2 is a half note, 4 is a quarter note, and so on. Zero is a 256th note.

wGridsPerBeat
Grids (subdivisions) per beat.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_IO_TIMESIG, **DMUS_TIMESIG_PMSG**

DMUS_IO_TOOL_HEADER

Contains information about a tool. Used in the Tool Form.

```
typedef struct _DMUS_IO_TOOL_HEADER {
    GUID    guidClassID;
    long    lIndex;
    DWORD   cPChannels;
    FOURCC  ckid;
    FOURCC  fccType;
    DWORD   dwPChannels[1];
} DMUS_IO_TOOL_HEADER;
```

Members

guidClassID
Class identifier of the tool.

lIndex
Position in the graph.

cPChannels
Number of items in the **dwPChannels** array.

ckid
Identifier of tool's data chunk. If this value is 0, it is assumed that the chunk is of type LIST, so **fccType** is valid and must be nonzero.

fccType

List type. If this value is 0, **ckid** is valid and must be nonzero.

dwPChannels
Array of performance channels for which the tool is valid.

Requirements

Header: Declared in dmusicf.h.

See Also

IDirectMusicGraph8::InsertTool

DMUS_IO_TRACK_EXTRAS_HEADER

Used in the Track Form.

```
typedef struct _DMUS_IO_TRACK_EXTRAS_HEADER {
    DWORD  dwFlags;
} DMUS_IO_TRACK_EXTRAS_HEADER;
```

Members

dwFlags
Flags for control tracks. For possible values, see **IDirectMusicSegment8::SetTrackConfig**.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_TRACK_HEADER

Contains information about a track. Used in the Track Form.

```
typedef struct _DMUS_IO_TRACK_HEADER {
    GUID     guidClassID;
    DWORD    dwPosition;
    DWORD    dwGroup;
    FOURCC   ckid;
    FOURCC   fccType;
} DMUS_IO_TRACK_HEADER;
```

Members

guidClassID
Class identifier of the track.

dwPosition
Position in the track list.

dwGroup
Group bits for the track.

ckid
Identifier of the track's data chunk. If this value is 0, it is assumed that the chunk is of type LIST, so **fccType** is valid and must be nonzero.

fccType
List type. If this value is 0, **ckid** is valid and must be nonzero.

Requirements

Header: Declared in dmusicf.h.

See Also

IDirectMusicSegment8::GetTrackGroup, **IDirectMusicSegment8::InsertTrack**, Track Form

DMUS_IO_TRACKREF_HEADER

Not implemented in DirectX 8.0.

Contains information about a track reference in a Song Form.

```
typedef struct _DMUS_IO_TRACKREF_HEADER {
    DWORD       dwSegmentID;
    DWORD       dwFlags;
} DMUS_IO_TRACKREF_HEADER;
```

Members

dwSegmentID
Identifier of the containing segment.

dwFlags
Reserved.

DMUS_IO_TRANSITION_DEF

Not implemented in DirectX 8.0.

Describes a transition. Used in the Song Form.

```
typedef struct _DMUS_IO_TRANSITION_DEF {
    DWORD  dwSegmentID;
    DWORD  dwTransitionID;
    DWORD  dwPlayFlags;
} DMUS_IO_TRANSITION_DEF;
```

Members

dwSegmentID
Segment after the transition, or one of the following constants:

DMUS_SONG_ANYSEG
Any segment.

DMUS_SONG_NOFROMSEG
No preceding segment; the transition is an intro.

DMUS_SONG_NOSEG
No following segment; the transition is an ending.

dwTransitionID
Template segment to use for the transition.

dwPlayFlags
Flags that control how the transition is played. See **DMUS_SEGF_FLAGS**.

Remarks

This structure describes a valid transition from another segment to this segment. When a song segment is played with the DMUS_SEGF_AUTOTRANSITION flag, a transition segment is composed and played. To find the appropriate transition segment, the performance looks at the currently playing segment and sees if it is a segment in the song. If it is, it gets the ID of that segment and then searches for it in the transition table of the destination segment. When it finds a match in **dwSourceSegment**, the performance composes **dwTransitionSegment** and plays it, using the **dwPlayFlags**.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_VALID_START

Contains information about a valid start point in a segment that is to be cued to a rhythm. Used in the Marker Track List.

```
typedef struct _DMUS_IO_VALID_START {
    MUSIC_TIME mtTime;
} DMUS_IO_VALID_START;
```

Members

mtTime
Time of the start point.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_VERSION

Contains the version number of the data. Used in the version subchunk of various chunks. See Common Chunks.

```
typedef struct _DMUS_IO_VERSION {
    DWORD  dwVersionMS;
    DWORD  dwVersionLS;
} DMUS_IO_VERSION;
```

Members

dwVersionMS
High-order 32 bits of the version number.

dwVersionLS
Low-order 32 bits of the version number.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_WAVE_HEADER

Describes streaming characteristics of a wave. Used in the Wave Header Chunk of a wave file.

```
typedef struct _DMUS_IO_WAVE_HEADER {
    REFERENCE_TIME  rtReadAhead;
    DWORD           dwFlags;
} DMUS_IO_WAVE_HEADER;
```

Members

rtReadAhead
Time to read ahead in a streaming wave.

dwFlags

Flags. Can be zero or the following value.

DMUS_WAVEF_STREAMING
The wave is a streaming wave.

Requirements

Header: Declared in dmusicf.h.

DMUS_IO_WAVE_ITEM_HEADER

Contains data for a wave sound in a Wave Track List.

```
typedef struct _DMUS_IO_WAVE_ITEM_HEADER {
    long          lVolume;
    long          lPitch;
    DWORD         dwVariations;
    REFERENCE_TIME rtTime;
    REFERENCE_TIME rtStartOffset;
    REFERENCE_TIME rtReserved;
    REFERENCE_TIME rtDuration;
    MUSIC_TIME     mtLogicalTime;
    DWORD         dwLoopStart;
    DWORD         dwLoopEnd;
    DWORD         dwFlags;
} DMUS_IO_WAVE_ITEM_HEADER;
```

Members

lVolume
Gain, in hundredths of a decibel. Must be a negative value.

lPitch
Pitch offset, in hundredths of a semitone.

dwVariations
Variation flags. One bit is set for each variation this wave belongs to.

rtTime
Start time, in reference time if the track is in clock time format; otherwise in music time.

rtStartOffset
Distance into wave to start playback, in reference time.

rtReserved
Not used.

rtDuration
Duration, in reference time if the track is in clock time format; otherwise in music time.

mtLogicalTime

Musical boundary where this belongs. Ignored if the track is in clock time format.

dwLoopStart
Start point for a looping wave.

dwLoopEnd
End point for a looping wave.

dwFlags
Flags. Can be 0 or one of the following values:

DMUS_WAVEF_NOINVALIDATE
This wave is not to be invalidated.

DMUS_WAVEF_STREAMING
Wave is streaming.

Requirements

Header: Declared in dmusicf.h.

Remarks

Logical Time vs. Actual Time

DMUS_IO_WAVE_PART_HEADER

Contains data for a Wave Track List.

```
typedef struct _DMUS_IO_WAVE_PART_HEADER {
    long   lVolume;
    DWORD  dwVariations;
    DWORD  dwPChannel;
    DWORD  dwLockToPart;
    DWORD  dwFlags;
    DWORD  dwIndex;
} DMUS_IO_WAVE_PART_HEADER;
```

Members

lVolume
Gain, in hundredths of a decibel, to apply to all waves in this wave part. This must be a negative value.

dwVariations
Active variations. One bit is set for each active variation.

dwPChannel
Performance channel of the part.

dwLockToPart

Variation lock identifier. Parts with the same value in this member always play the same variation. A value of 0 means that the part plays its variations independently of all other parts.

dwFlags
Flags for managing how variations are chosen, in the lower four bits. See **DMUS_VARIATIONT_TYPES**.

dwIndex
Index for distinguishing multiple parts on the same performance channel.

DMUS_IO_WAVE_TRACK_HEADER

Contains data for a wave track in a Wave Track List.

```
typedef struct _DMUS_IO_WAVE_TRACK_HEADER {
    long   lVolume;
    DWORD  dwFlags;
} DMUS_IO_WAVE_TRACK_HEADER;
```

Members

lVolume
Gain, hundredths of a decibel, to be applied to all waves.

dwFlags
Flags. Can be 0 or one or more of the following values:

DMUS_WAVETRACKF_SYNC_VAR
The track gets its variations from a pattern track. For more information, see GUID_Variations.

DMUS_WAVETRACKF_PERSIST_CONTROL
Variation control information persists from one playback instance to the next.

Requirements

Header: Declared in dmusicf.h.

DSOUND_IO_3D

Contains 3-D parameters for a DirectSound buffer in a DirectSound Buffer Configuration Form.

```
typedef struct _DSOUND_IO_3D {
    GUID guid3DAlgorithm;
    DS3DBUFFER ds3d;
} DSOUND_IO_3D;
```

Members

guid3DAlgorithm
Unique identifier of the 3-D algorithm to use. See **DSBUFFERDESC**.

ds3d
DS3DBUFFER structure that contains the parameters. This information is valid only if DSBCAPS_CTRL3D is set in the buffer description.

Requirements

Header: Declared in dmusicf.h.

DSOUND_IO_DSBUFFERDESC

Describes a DirectSound buffer. Used in the DirectSound Buffer Configuration Form.

```
typedef struct _DSOUND_IO_DSBUFFERDESC {
    DWORD dwFlags;
    WORD nChannels;
    LONG lVolume;
    LONG lPan;
    DWORD dwReserved;
} DSOUND_IO_DSBUFFERDESC;
```

Members

dwFlags
Buffer creation flags. See **DSBUFFERDESC**.

nChannels
Number of channels. Other parameters of the format are determined by the sink that owns the buffer.

lVolume
Initial volume. Used only if DSBCAPS_CTRLVOLUME is in **dwFlags**.

lPan
Initial pan. Used only if DSBCAPS_CTRLPAN is in **dwFlags**.

dwReserved
Reserved. Must be 0.

Requirements

Header: Declared in dmusicf.h.

DSOUND_IO_DSBUSID

Contains bus identifiers. Used in the DirectSound Buffer Configuration Form.

```
typedef struct _DSOUND_IO_DSBUSID {
    DWORD busid[1];
} DSOUND_IO_DSBUSID;
```

Members

busid

Array of **DWORD**s containing the bus identifiers. The size of the array can be determined from the chunk size.

Requirements

Header: Declared in dmusicf.h.

DSOUND_IO_DXDMO_DATA

Contains data for a DMO.

```
typedef struct _DSOUND_IO_DXDMO_DATA {
    DWORD data[1];
} DSOUND_IO_DXDMO_DATA;
```

Members

data

Array of **DWORD** values containing the data. The size of the array can be determined from the chunk size.

Requirements

Header: Declared in dmusicf.h.

DSOUND_IO_DXDMO_HEADER

Contains header information for a DMO chunk in an Effects Form.

```
typedef struct _DSOUND_IO_DXDMO_HEADER {
    DWORD dwEffectFlags;
    GUID guidDSFXClass;
    GUID guidReserved;
    GUID guidSendBuffer;
    DWORD dwReserved;
} DSOUND_IO_DXDMO_HEADER;
```

Members

dwEffectFlags
Effect creation flags. See **DSEFFECTDESC**.

guidDSFXClass
Class identifier of the effect.

guidReserved
Reserved. Must be GUID_NULL.

guidSendBuffer
Unique identifier of the buffer to send to, if this is a send effect.

dwReserved
Reserved. Must be 0.

Requirements

Header: Declared in dmusicf.h.

Standard Track Parameters

This section describes the standard track parameters that can be set and retrieved by using the following methods.

- **IDirectMusicPerformance8::GetParam**
- **IDirectMusicPerformance8::GetParamEx**
- **IDirectMusicPerformance8::SetParam**
- **IDirectMusicSegment8::GetParam**
- **IDirectMusicSegment8::SetParam**
- **IDirectMusicTrack8::GetParam**
- **IDirectMusicTrack8::GetParamEx**
- **IDirectMusicTrack8::SetParam**
- **IDirectMusicTrack8::SetParamEx**

Parameter types are listed in this section under their GUIDs, as specified in the *rGuidType* parameter of the method call.

The following information is given for each parameter type:

Element	Description
Track type	Tracks to which the parameters apply. See Standard Track Types. Although parameters are always associated with particular tracks, applications usually call the method on the segment or the

	performance and let DirectMusic find the appropriate track. See Identifying the Track.
Data type	Type of data pointed to by the *pParam* parameter of the method call by which the parameter is set or retrieved.
mtTime	Significance, if any, of the *mtTime* parameter of the method call by which the parameter is set or retrieved.

Parameter types are as follows:

- GUID_BandParam
- GUID_ChordParam
- GUID_Clear_All_Bands
- GUID_Clear_All_MelodyFragments
- GUID_CommandParam
- GUID_CommandParam2
- GUID_CommandParamNext
- GUID_ConnectToDLSCollection
- GUID_Disable_Auto_Download
- GUID_DisableTempo
- GUID_DisableTimeSig
- GUID_Download• GUID_DownloadToAudioPath
- GUID_Enable_Auto_Download
- GUID_EnableTempo
- GUID_EnableTimeSig
- GUID_IDirectMusicBand
- GUID_IDirectMusicChordMap
- GUID_IDirectMusicStyle
- GUID_MelodyFragment
- GUID_MuteParam
- GUID_Play_Marker
- GUID_RhythmParam
- GUID_SeedVariations
- GUID_StandardMIDIFile
- GUID_TempoParam
- GUID_TimeSignature
- GUID_Unload
- GUID_UnloadFromAudioPath
- GUID_Valid_Start_Time
- GUID_Variations

GUID_BandParam

Sets or retrieves a band.

Track type	Band
Data type (**pParam***)**	**DMUS_BAND_PARAM**
mtTime	The logical time at which to set the band, or the time for which to retrieve the band

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_ChordParam

Sets or retrieves a chord change.

Track type	Chord
Data type (**pParam***)**	**DMUS_CHORD_PARAM**
mtTime	The time, in track time, at which to add the chord to the track, or the time at or directly after the chord to be retrieved from the track

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_Clear_All_Bands

Clears all bands from the track. This parameter can be set but not retrieved.

Track type	Band
Data type (**pParam***)**	None
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_Clear_All_MelodyFragments

Not implemented in DirectX 8.0.

Clears all melody fragments from the track. This parameter can be set but not retrieved.

Track type	Melody fragment
Data type (**pParam***)**	None
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_CommandParam

Sets or retrieves a groove or embellishment command.

Track type	Command
Data type (**pParam***)**	**DMUS_COMMAND_PARAM_2**
mtTime	The time, in track time, at which to add the command to the track, or the time at or directly after the command to be retrieved from the track

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_CommandParam2

Sets or retrieves a groove or embellishment command.

Track type	Command
Data type (**pParam***)**	**DMUS_COMMAND_PARAM_2**. The **mtTime** member of this structure gives the actual time of the command.
mtTime	The time, in track time, at which to add the command to the track, or the time at or directly after the command to be retrieved from the track

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_CommandParamNext

Sets or retrieves a groove or embellishment command to follow all other commands in the command track.

Track type	Command
Data type (**pParam***)**	**DMUS_COMMAND_PARAM_2**
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_ConnectToDLSCollection

Connects all bands in the track to a DLS collection. This parameter can be set but not retrieved.

Track type	Band
Data type (**pParam***)**	**IDirectMusicCollection8** pointer
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

Playing a MIDI File with Custom Instruments.

GUID_Disable_Auto_Download

Disables automatic downloading of instruments and waves. This parameter can be set but not retrieved.

Track type	Band, wave
Data type (**pParam***)**	None
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

Using Bands, GUID_Enable_Auto_Download

GUID_DisableTempo

Disables tempo messages. This parameter can be set but not retrieved.

Track type	Tempo
Data type (**pParam***)**	None
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

Disabling and Enabling Track Parameters, GUID_EnableTempo

GUID_DisableTimeSig

Disables time signature messages. This parameter can be set but not retrieved.

Track type	Pattern, time signature, style, motif
Data type (**pParam***)**	None
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

Disabling and Enabling Track Parameters, GUID_EnableTimeSig

GUID_Download

Downloads instrument data or wave data. This parameter can be set but not retrieved.

Track type	Band, wave
Data type (**pParam***)**	**IDirectMusicPerformance8** pointer
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::Download

GUID_DownloadToAudioPath

Downloads instrument data or wave data. This parameter can be set but not retrieved.

Track type	Band, wave
Data type (**pParam***)**	**IDirectMusicAudioPath8** pointer
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment8::Download

GUID_Enable_Auto_Download

Enables automatic downloading of instruments and waves. This parameter can be set but not retrieved.

Track type	Band, wave
Data type (**pParam***)**	None
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

Using Bands, GUID_Disable_Auto_Download

GUID_EnableTempo

Enables tempo messages. This parameter can be set but not retrieved.

Track type	Tempo
Data type (**pParam***)**	None
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

Disabling and Enabling Track Parameters, GUID_DisableTempo

GUID_EnableTimeSig

Enables time signature messages. This parameter can be set but not retrieved.

Track type	Pattern, time signature, style, motif
Data type (**pParam***)**	None
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

Disabling and Enabling Track Parameters, GUID_DisableTimeSig

GUID_IDirectMusicBand

Sets a band. This parameter can be set but not retrieved.

Track type	Band
Data type (**pParam***)**	**IDirectMusicBand8**

mtTime	The time, in track time, at which to add the band to the track

Remarks

For an explanation of the table, see Standard Track Parameters.

This parameter has been superseded by GUID_BandParam, which allows you to specify the physical time.

Requirements

Header: Declared in dmusici.h.

GUID_IDirectMusicChordMap

Sets or retrieves the chordmap.

Track type	Chordmap
Data type (**pParam***)**	**IDirectMusicChordMap8** pointer or address of a variable to receive this pointer
mtTime	The time, in track time, at which to add the chordmap to the track, or the time at or directly after the chordmap to be retrieved from the track

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_IDirectMusicStyle

Sets or retrieves the style.

Track type	Style
Data type (**pParam***)**	**IDirectMusicStyle8** pointer or address of a variable to receive this pointer
mtTime	The time, in track time, at which to add the style to the track, or the time at or directly after the style to be retrieved from the track

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_MelodyFragment

Not implemented in DirectX 8.0.

Sets or retrieves a melody fragment.

Track type	Melody fragment
Data type (**pParam***)**	**DMUS_MELODY_FRAGMENT**
mtTime	The time, in track time, at which to add the fragment to the track, or the time at or directly after the fragment to retrieve from the track

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_MuteParam

Sets or retrieves channel-mapping information.

Track type	Mute
Data type (**pParam***)**	**DMUS_MUTE_PARAM**. The **dwPChannel** member must be initialized before this structure is passed to the get method.
mtTime	The time, in track time, at which to add the mute event to the track, or the time at or directly after the mute event to be retrieved from the track

Remarks

For an explanation of the table, see Standard Track Parameters.

You cannot dynamically change the mapping or muting of a channel by setting this parameter while the mute track is playing. Parameters in the mute track are retrieved by internal **GetParam** calls from other tracks, and the application has no control over

the timing of such calls. Changes in the mapping or muting of channels should be authored into the mute track of a segment played as a controlling segment, or done at run time by tools.

Requirements

Header: Declared in dmusici.h.

GUID_Play_Marker

Retrieves the next point in the currently playing segment at which a new segment can start. This parameter can be retrieved but not set.

Track type	Marker
Data type (**pParam***)**	**DMUS_PLAY_MARKER_PARAM**
mtTime	Track time at which to start seeking a marker

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

See Also

Segment Timing

GUID_RhythmParam

Retrieves the rhythm pattern for a sequence of chords stored in a measure in the track. This parameter can be retrieved but not set.

Track type	Chord
Data type (**pParam***)**	**DMUS_RHYTHM_PARAM**. The **TimeSig** member must be initialized before this structure is passed to the get method.
mtTime	The time, in track time, at or directly after the beginning of the measure containing the rhythm pattern to be retrieved from the track

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_SeedVariations

Seeds the random number generator for variation selection. A nonzero value is used as the seed. A value of 0 resets the default behavior of getting the seed from the system clock. This parameter can be set but not retrieved.

Track type	Pattern, style, motif
Data type (**pParam***)**	**Long**
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

Setting this parameter to nonzero is useful for testing because it ensures that the same sequence of random numbers is generated each time. The parameter should be set only once, before the track is played. The style and command track must be designed so that each time that the segment is played, the same patterns are chosen at the same places in the segment. Each loop plays different variations than the one before it does, but each time the entire segment is replayed from the beginning, each loop sounds the same as the first time the segment was played.

Requirements

Header: Declared in dmusici.h.

GUID_StandardMIDIFile

Ensures that a standard MIDI file (one not authored specifically for DirectMusic) plays correctly. This parameter can be set but not retrieved.

Track type	Band
Data type (**pParam***)**	None
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

This parameter must be set for any segment based on a standard MIDI file before any instruments are downloaded.

Requirements

Header: Declared in dmusici.h.

GUID_TempoParam

Sets or retrieves the tempo.

Track type	Tempo
Data type (**pParam***)**	**DMUS_TEMPO_PARAM**. When setting the parameter, the **mtTime** member of the structure is ignored. When getting the parameter, the **mtTime** member receives the offset of the tempo change from the requested time and is always 0 or less.
mtTime	The time, in track time, at which to set the tempo, or the time at or directly after the tempo change to retrieve

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_TimeSignature

Obtains the time signature. This parameter can be retrieved but not set.

Track type	Time signature and style
Data type (**pParam***)**	**DMUS_TIMESIGNATURE**. The **mtTime** member receives the offset of the time signature change from the requested time and is always 0 or less.
mtTime	The time, in track time, at which to set the time signature, or the time at or directly after the time signature change to retrieve

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

GUID_Unload

Unloads instrument or wave data. This parameter can be set but not retrieved.

Track type	Band, wave
Data type (**pParam***)**	**IDirectMusicPerformance8** pointer
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

See Also

IDirectMusicSegment8::Unload

Requirements

Header: Declared in dmusici.h.

GUID_UnloadFromAudioPath

Unloads instrument or wave data. This parameter can be set but not retrieved.

Track type	Band, wave
Data type (**pParam***)**	**IDirectMusicAudioPath8** pointer
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

See Also

IDirectMusicSegment8::Unload

Requirements

Header: Declared in dmusici.h.

GUID_Valid_Start_Time

Obtains the next valid point within a segment at which it can start. This parameter can be retrieved but not set.

Track type	Marker, motif, pattern

Data type (**pParam***)**	**DMUS_VALID_START_PARAM**
mtTime	Not used

Remarks

For an explanation of the table, see Standard Track Parameters.

See Also

Segment Timing

Requirements

Header: Declared in dmusici.h.

GUID_Variations

Obtains the variations in effect across performance channels. This parameter can be retrieved but not set.

Track type	Pattern
Data type (**pParam***)**	**DMUS_VARIATIONS_PARAM**
mtTime	Not used.

Remarks

For an explanation of the table, see Standard Track Parameters.

Requirements

Header: Declared in dmusici.h.

DirectMusic Enumerated Types

This section contains references for the following enumerated types:

- **DMUS_CHORDKEYF_FLAGS**
- **DMUS_CLOCKTYPE**
- **DMUS_COMMANDT_TYPES**
- **DMUS_COMPOSEF_FLAGS**
- **DMUS_CURVE_FLAGS**

- **DMUS_NOTEF_FLAGS**
- **DMUS_PATTERNT_TYPES**
- **DMUS_PLAYMODE_FLAGS**
- **DMUS_PMSGF_FLAGS**
- **DMUS_PMSGT_TYPES**
- **DMUS_SEGF_FLAGS**
- **DMUS_SHAPET_TYPES**
- **DMUS_STYLET_TYPES**
- **DMUS_TIME_RESOLVE_FLAGS**
- **DMUS_TRACKF_FLAGS**
- **DMUS_VARIATIONT_TYPES**

DMUS_CHORDKEYF_FLAGS

Used in the **bFlags** member of the **DMUS_CHORD_KEY** structure.

```
typedef enum enumDMUS_CHORDKEYF_FLAGS {
    DMUS_CHORDKEYF_SILENT  = 1,
} DMUS_CHORDKEYF_FLAGS;
```

Constants

DMUS_CHORDKEYF_SILENT
The chord is silent.

Requirements

Header: Declared in dmusici.h.

DMUS_CLOCKTYPE

Used in the **ctType** member of the **DMUS_CLOCKINFO8** structure.

```
typedef enum {
    DMUS_CLOCK_SYSTEM = 0,
    DMUS_CLOCK_WAVE   = 1
} DMUS_CLOCKTYPE;
```

Constants

DMUS_CLOCK_SYSTEM
Clock is the system clock.

DMUS_CLOCK_WAVE

Clock is on a wave-playback device.

Requirements

Header: Declared in dmusicc.h.

DMUS_COMMANDT_TYPES

Used in the *wCommand* parameter of the **IDirectMusicComposer8::AutoTransition** and **IDirectMusicComposer8::ComposeTransition** methods and in the **bCommand** member of the **DMUS_COMMAND_PARAM** structure.

```
typedef enum enumDMUS_COMMANDT_TYPES {
    DMUS_COMMANDT_GROOVE      = 0,
    DMUS_COMMANDT_FILL        = 1,
    DMUS_COMMANDT_INTRO       = 2,
    DMUS_COMMANDT_BREAK       = 3,
    DMUS_COMMANDT_END         = 4,
    DMUS_COMMANDT_ENDANDINTRO = 5
} DMUS_COMMANDT_TYPES;
```

Constants

DMUS_COMMANDT_GROOVE
: The command is a groove command.

DMUS_COMMANDT_FILL
: The command is a fill.

DMUS_COMMANDT_INTRO
: The command is an introduction.

DMUS_COMMANDT_BREAK
: The command is a break.

DMUS_COMMANDT_END
: The command is an ending.

DMUS_COMMANDT_ENDANDINTRO
: The command is an ending and an introduction.

Requirements

Header: Declared in dmusici.h.

DMUS_COMPOSEF_FLAGS

Used in the *dwFlags* parameter of the **IDirectMusicComposer8::ComposeTransition** and **IDirectMusicComposer8::AutoTransition** methods.

```
typedef enum enumDMUS_COMPOSEF_FLAGS {
    DMUS_COMPOSEF_NONE              =       0,
    DMUS_COMPOSEF_ALIGN             =     0x1,
    DMUS_COMPOSEF_OVERLAP           =     0x2,
    DMUS_COMPOSEF_IMMEDIATE         =     0x4,
    DMUS_COMPOSEF_GRID              =     0x8,
    DMUS_COMPOSEF_BEAT              =    0x10,
    DMUS_COMPOSEF_MEASURE           =    0x20,
    DMUS_COMPOSEF_AFTERPREPARETIME  =    0x40,
    DMUS_COMPOSEF_VALID_START_BEAT  =    0x80,
    DMUS_COMPOSEF_VALID_START_GRID  =   0x100,
    DMUS_COMPOSEF_VALID_START_TICK  =   0x200,
    DMUS_COMPOSEF_SEGMENTEND        =   0x400,
    DMUS_COMPOSEF_MARKER            =   0x800,
    DMUS_COMPOSEF_MODULATE          =  0x1000,
    DMUS_COMPOSEF_LONG              =  0x2000,
    DMUS_COMPOSEF_ENTIRE_TRANSITION =  0x4000,
    DMUS_COMPOSEF_1BAR_TRANSITION   =  0x8000,
    DMUS_COMPOSEF_ENTIRE_ADDITION   = 0x10000,
    DMUS_COMPOSEF_1BAR_ADDITION     = 0x20000,
    DMUS_COMPOSEF_VALID_START_MEASURE = 0x40000,
    DMUS_COMPOSEF_DEFAULT           = 0x80000,
    DMUS_COMPOSEF_NOINVALIDATE      = 0x100000,
    DMUS_COMPOSEF_USE_AUDIOPATH     = 0x200000
} DMUS_COMPOSEF_FLAGS;
```

Constants

DMUS_COMPOSEF_NONE
: No flags. By default, the transition starts on a measure boundary.

DMUS_COMPOSEF_ALIGN
: Align transition to the time signature of the currently playing segment.

DMUS_COMPOSEF_OVERLAP
: Overlap the transition into *pToSeg*. This flag is not currently implemented.

DMUS_COMPOSEF_IMMEDIATE
: **AutoTransition** only. Start transition on a music or a reference-time boundary.

DMUS_COMPOSEF_GRID
: **AutoTransition** only. Start transition on a grid boundary.

DMUS_COMPOSEF_BEAT

AutoTransition only. Start transition on a beat boundary.

DMUS_COMPOSEF_MEASURE
AutoTransition only. Start transition on a measure boundary.

DMUS_COMPOSEF_AFTERPREPARETIME
AutoTransition only. Use the DMUS_SEGF_AFTERPREPARETIME flag when cueing the transition.

DMUS_COMPOSEF_VALID_START_BEAT
Allow the switch to occur on any beat. Used in conjunction with DMUS_COMPOSEF_ALIGN.

DMUS_COMPOSEF_VALID_START_GRID
Allow the switch to occur on any grid. Used in conjunction with DMUS_COMPOSEF_ALIGN.

DMUS_COMPOSEF_VALID_START_TICK
Allow the switch to occur at any time. Used in conjunction with DMUS_COMPOSEF_ALIGN.

DMUS_COMPOSEF_SEGMENTEND
Play the transition at the end of the current segment.

DMUS_COMPOSEF_MARKER
Play the transition at the next marker in the current segment.

DMUS_COMPOSEF_MODULATE
Compose a transition that modulates smoothly from *pFromSeg* to *pToSeg*, using the chord of *pToSeg*.

DMUS_COMPOSEF_LONG
Composes a long transition. If this flag is not set, the length of the transition is at most one measure unless the *wCommand* parameter of **ComposeTransition** or **AutoTransition** specifies an ending and the style contains an ending of greater than one measure. If this flag is set, the length of the transition increases by one measure.

DMUS_COMPOSEF_ENTIRE_TRANSITION
Include the entire transition pattern.

DMUS_COMPOSEF_1BAR_TRANSITION
Include one bar of the transition pattern.

DMUS_COMPOSEF_ENTIRE_ADDITION
Include the additional transition pattern in its entirety. Used in combination with DMUS_COMPOSEF_LONG.

DMUS_COMPOSEF_1BAR_ADDITION
Include one bar of the additional transition pattern. This is the default behavior when DMUS_COMPOSEF_LONG is specified.

DMUS_COMPOSEF_VALID_START_MEASURE
Allow the switch to occur on any bar. Used in combination with DMUS_COMPOSEF_ALIGN.

DMUS_COMPOSEF_DEFAULT
Use the segment's default boundary.

DMUS_COMPOSEF_NOINVALIDATE
Do not invalidate segments that are playing.

DMUS_COMPOSEF_USE_AUDIOPATH
Use the audiopaths embedded in the segments.

Requirements

Header: Declared in dmusici.h.

See Also

DMUS_SEGF_FLAGS

DMUS_CURVE_FLAGS

Used in the **bFlags** member of the **DMUS_CURVE_PMSG** structure.

```
typedef enum enumDMUS_CURVE_FLAGS {
    DMUS_CURVE_RESET            = 1,
    DMUS_CURVE_START_FROM_CURRENT = 2,
} DMUS_CURVE_FLAGS;
```

Constants

DMUS_CURVE_RESET
The value of **DMUS_CURVE_PMSG.nResetValue** must be set when the time is reached or an invalidation occurs because of a transition. If this flag is not set, the curve stays permanently at the new value.

DMUS_CURVE_START_FROM_CURRENT
Ignore **DMUS_CURVE_PMSG.nStartValue** and start the curve at the current value. This works only for volume, expression, and pitch bend.

Requirements

Header: Declared in dmusici.h.

DMUS_NOTEF_FLAGS

Used in the **bFlags** member of the **DMUS_NOTE_PMSG** structure.

```
typedef enum enumDMUS_NOTEF_FLAGS {
    DMUS_NOTEF_NOTEON =             1,
    DMUS_NOTEF_NOINVALIDATE =       2,
    DMUS_NOTEF_NOINVALIDATE_INSCALE = 4,
    DMUS_NOTEF_NOINVALIDATE_INCHORD = 8,
```

```
    DMUS_NOTEF_REGENERATE =        0x10,
} DMUS_NOTEF_FLAGS;
```

Constants

DMUS_NOTEF_NOTEON
: MIDI note-on. When a **DMUS_NOTE_PMSG** is first sent by the **IDirectMusicPerformance8::SendPMsg** method, this flag should be set. If the flag is not set, the message is a note-off.

DMUS_NOTEF_NOINVALIDATE
: Do not invalidate the note.

DMUS_NOTEF_NOINVALIDATE_INSCALE
: Do not invalidate if the note is still within the scale.

DMUS_NOTEF_NOINVALIDATE_INCHORD
: Do not invalidate if the note is still within the chord.

DMUS_NOTEF_REGENERATE
: Regenerate the note when a chord change occurs. The note's music value, subchord level, and play mode flags are used to construct a new note according to the new chord and scale. If the original note had a timing offset, this is applied to the start time of the new note.

Remarks

The NOINVALIDATE flags ensure that the note plays for its full duration even when messages are invalidated.

Requirements

Header: Declared in dmusici.h.

DMUS_PATTERNT_TYPES

Used in various command structures to control the way patterns are selected in sequential commands.

```
typedef enum enumDMUS_PATTERNT_TYPES {
    DMUS_PATTERNT_RANDOM          = 0,
    DMUS_PATTERNT_REPEAT          = 1,
    DMUS_PATTERNT_SEQUENTIAL      = 2,
    DMUS_PATTERNT_RANDOM_START    = 3,
    DMUS_PATTERNT_NO_REPEAT       = 4,
    DMUS_PATTERNT_RANDOM_ROW      = 5
} DMUS_PATTERNT_TYPES;
```

Constants

DMUS_PATTERNT_RANDOM
Select a random matching pattern. This is the behavior in versions prior to DirectX 8.0.

DMUS_PATTERNT_REPEAT
Repeat the last matching pattern.

DMUS_PATTERNT_SEQUENTIAL
Play matching patterns sequentially, in the order loaded, starting with the first.

DMUS_PATTERNT_RANDOM_START
Play matching patterns sequentially, in the order loaded, starting at a random point in the sequence.

DMUS_PATTERNT_NO_REPEAT
Play randomly, but do not play the same pattern twice.

DMUS_PATTERNT_RANDOM_ROW
Play randomly, but do not repeat any pattern until all have played.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_COMMAND_PARAM, **DMUS_COMMAND_PARAM_2**, **DMUS_IO_COMMAND**, **DMUS_VARIATIONT_TYPES**

DMUS_PLAYMODE_FLAGS

Used in various structures for the basic play modes. The play mode determines how a music value is transposed to a MIDI note.

```
typedef enum enumDMUS_PLAYMODE_FLAGS {
    DMUS_PLAYMODE_KEY_ROOT          = 1,
    DMUS_PLAYMODE_CHORD_ROOT        = 2,
    DMUS_PLAYMODE_SCALE_INTERVALS   = 4,
    DMUS_PLAYMODE_CHORD_INTERVALS   = 8,
    DMUS_PLAYMODE_NONE              = 16,
} DMUS_PLAYMODE_FLAGS;
```

Constants

DMUS_PLAYMODE_KEY_ROOT
Transpose over the key root.

DMUS_PLAYMODE_CHORD_ROOT
Transpose over the chord root.

DMUS_PLAYMODE_SCALE_INTERVALS
Use scale intervals from a scale pattern.

DMUS_PLAYMODE_CHORD_INTERVALS
Use chord intervals from a chord pattern.

DMUS_PLAYMODE_NONE
No mode. Indicates that the parent part's mode should be used.

Remarks

The following defined values represent combinations of play mode flags:

DMUS_PLAYMODE_ALWAYSPLAY
Combination of DMUS_PLAYMODE_SCALE_INTERVALS, DMUS_PLAYMODE_CHORD_INTERVALS, and DMUS_PLAYMODE_CHORD_ROOT. If it is desirable to play a note that is above the top of the chord, this mode finds a position for the note by using intervals from the scale. Essentially, this mode is a combination of the normal and melodic playback modes, in which a failure in normal mode causes a second try in melodic mode.

DMUS_PLAYMODE_FIXED
Interpret the music value as a MIDI value. This is defined as 0 and signifies the absence of other flags. This flag is used for drums, sound effects, and sequenced notes that should not be transposed by the chord or scale.

DMUS_PLAYMODE_FIXEDTOCHORD
Same as DMUS_PLAYMODE_CHORD_ROOT. The music value is a fixed MIDI value, but it is transposed over the chord root.

DMUS_PLAYMODE_FIXEDTOKEY
Same as DMUS_PLAYMODE_KEY_ROOT. The music value is a fixed MIDI value, but it is transposed over the key root.

DMUS_PLAYMODE_MELODIC
Combination of DMUS_PLAYMODE_CHORD_ROOT and DMUS_PLAYMODE_SCALE_INTERVALS. The chord root is used, but the notes track only the intervals in the scale. The key root and chord intervals are ignored. This is useful for melodic lines that play relative to the chord root.

DMUS_PLAYMODE_NORMALCHORD
Combination of DMUS_PLAYMODE_CHORD_ROOT and DMUS_PLAYMODE_CHORD_INTERVALS. This is the prevalent playback mode. The notes track the intervals in the chord, which is based on the chord root. If the music value has a scale component, the additional intervals are pulled from the scale and added. If the chord does not have an interval to match the chord component of the music value, the note is silent.

DMUS_PLAYMODE_PEDALPOINT
Combination of DMUS_PLAYMODE_KEY_ROOT and DMUS_PLAYMODE_SCALE_INTERVALS. The key root is used, and the

notes track only the intervals in the scale. The chord root and intervals are ignored. This is useful for melodic lines that play relative to the key root.

DMUS_PLAYMODE_PEDALPOINTALWAYS
Combination of DMUS_PLAYMODE_PEDALPOINT and DMUS_PLAYMODE_PEDALPOINTCHORD. Chord intervals are used if possible; otherwise scale intervals are used.

DMUS_PLAYMODE_PEDALPOINTCHORD
Combination of DMUS_PLAYMODE_MELODIC and DMUS_PLAYMODE_NORMALCHORD. The key root is used and the notes track only the intervals in the chord. The chord root and scale intervals are completely ignored. This is useful for chordal lines that play relative to the key root.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::MIDIToMusic, **IDirectMusicPerformance8::MusicToMIDI**, **DMUS_NOTE_PMSG**, **DMUS_IO_STYLENOTE**, **DMUS_IO_STYLEPART**, Music Values and MIDI Notes

DMUS_PMSGF_FLAGS

Used in the **dwFlags** member of the **DMUS_PMSG** structure.

```
typedef enum enumDMUS_PMSGF_FLAGS {
    DMUS_PMSGF_REFTIME        = 1,
    DMUS_PMSGF_MUSICTIME      = 2,
    DMUS_PMSGF_TOOL_IMMEDIATE = 4,
    DMUS_PMSGF_TOOL_QUEUE     = 8,
    DMUS_PMSGF_TOOL_ATTIME    = 0x10,
    DMUS_PMSGF_TOOL_FLUSH     = 0x20,
    DMUS_PMSGF_LOCKTOREFTIME  = 0x40,
    DMUS_PMSGF_DX8            = 0x80
} DMUS_PMSGF_FLAGS;
```

Constants

DMUS_PMSGF_REFTIME
The **rtTime** member is valid.

DMUS_PMSGF_MUSICTIME
The **mtTime** member is valid.

DMUS_PMSGF_TOOL_IMMEDIATE
Message should be processed immediately, regardless of its time stamp.

DMUS_PMSGF_TOOL_QUEUE
Message should be processed just before its time stamp, allowing for port latency.

DMUS_PMSGF_TOOL_ATTIME
Message should be processed at the time stamp.

DMUS_PMSGF_TOOL_FLUSH
Message is being flushed.

DMUS_PMSGF_LOCKTOREFTIME
Value in **rtTime** cannot be overridden by a tempo change.

DMUS_PMSGF_DX8
Message has valid members not present in versions prior to DirectX 8.0.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::SendPMsg,
IDirectMusicTool8::GetMsgDeliveryType

DMUS_PMSGT_TYPES

Used in the **dwType** member of the **DMUS_PMSG** structure to identify the type of message.

```
typedef enum enumDMUS_PMSGT_TYPES {
    DMUS_PMSGT_MIDI           = 0,
    DMUS_PMSGT_NOTE           = 1,
    DMUS_PMSGT_SYSEX          = 2,
    DMUS_PMSGT_NOTIFICATION   = 3,
    DMUS_PMSGT_TEMPO          = 4,
    DMUS_PMSGT_CURVE          = 5,
    DMUS_PMSGT_TIMESIG        = 6,
    DMUS_PMSGT_PATCH          = 7,
    DMUS_PMSGT_TRANSPOSE      = 8,
    DMUS_PMSGT_CHANNEL_PRIORITY = 9,
    DMUS_PMSGT_STOP           = 10,
    DMUS_PMSGT_DIRTY          = 11,
    DMUS_PMSGT_WAVE           = 12,
    DMUS_PMSGT_LYRIC          = 13,
    DMUS_PMSGT_SCRIPTLYRIC    = 14,
    DMUS_PMSGT_USER           = 255
```

} DMUS_PMSGT_TYPES;

Constants

DMUS_PMSGT_MIDI
MIDI channel message. See **DMUS_MIDI_PMSG**.

DMUS_PMSGT_NOTE
Music note. See **DMUS_NOTE_PMSG**.

DMUS_PMSGT_SYSEX
MIDI system exclusive message. See **DMUS_SYSEX_PMSG**.

DMUS_PMSGT_NOTIFICATION
Notification message. See **DMUS_NOTIFICATION_PMSG**.

DMUS_PMSGT_TEMPO
Tempo message. See **DMUS_TEMPO_PMSG**.

DMUS_PMSGT_CURVE
Control change and pitch-bend curve. See **DMUS_CURVE_PMSG**.

DMUS_PMSGT_TIMESIG
Time signature. See **DMUS_TIMESIG_PMSG**.

DMUS_PMSGT_PATCH
Patch change. See **DMUS_PATCH_PMSG**.

DMUS_PMSGT_TRANSPOSE
Transposition. See **DMUS_TRANSPOSE_PMSG**.

DMUS_PMSGT_CHANNEL_PRIORITY
Channel priority change. See **DMUS_CHANNEL_PRIORITY_PMSG**.

DMUS_PMSGT_STOP
Stop message. See **DMUS_PMSG**.

DMUS_PMSGT_DIRTY
A control segment has started or ended. See **DMUS_PMSG**.

DMUS_PMSGT_WAVE
Control information for playing a wave. See **DMUS_WAVE_PMSG**.

DMUS_PMSGT_LYRIC
Lyric message. See **DMUS_LYRIC_PMSG**.

DMUS_PMSGT_SCRIPTLYRIC
Lyric message sent by a script. See **DMUS_LYRIC_PMSG**.

DMUS_PMSGT_USER
User-defined message.

Requirements

Header: Declared in dmusici.h.

DMUS_SEGF_FLAGS

Passed to various methods of **IDirectMusicPerformance8** to control the timing and other aspects of actions on a segment.

```
typedef enum enumDMUS_SEGF_FLAGS {
    DMUS_SEGF_REFTIME            = 1<<6,
    DMUS_SEGF_SECONDARY          = 1<<7,
    DMUS_SEGF_QUEUE              = 1<<8,
    DMUS_SEGF_CONTROL            = 1<<9,
    DMUS_SEGF_AFTERPREPARETIME   = 1<<10,
    DMUS_SEGF_GRID               = 1<<11,
    DMUS_SEGF_BEAT               = 1<<12,
    DMUS_SEGF_MEASURE            = 1<<13,
    DMUS_SEGF_DEFAULT            = 1<<14,
    DMUS_SEGF_NOINVALIDATE       = 1<<15,
    DMUS_SEGF_ALIGN              = 1<<16,
    DMUS_SEGF_VALID_START_BEAT   = 1<<17,
    DMUS_SEGF_VALID_START_GRID   = 1<<18,
    DMUS_SEGF_VALID_START_TICK   = 1<<19,
    DMUS_SEGF_AUTOTRANSITION     = 1<<20,
    DMUS_SEGF_AFTERQUEUETIME     = 1<<21,
    DMUS_SEGF_AFTERLATENCYTIME   = 1<<22,
    DMUS_SEGF_SEGMENTEND         = 1<<23,
    DMUS_SEGF_MARKER             = 1<<24,
    DMUS_SEGF_TIMESIG_ALWAYS     = 1<<25,
    DMUS_SEGF_USE_AUDIOPATH      = 1<<26,
    DMUS_SEGF_VALID_START_MEASURE = 1<<27
} DMUS_SEGF_FLAGS;
```

Constants

DMUS_SEGF_REFTIME
: Time parameter is in reference time.

DMUS_SEGF_SECONDARY
: Secondary segment.

DMUS_SEGF_QUEUE
: Put at the end of the primary segment queue. Valid for primary segments only.

DMUS_SEGF_CONTROL
: Play as a control segment. Valid for secondary segments only. See Remarks.

DMUS_SEGF_AFTERPREPARETIME
: Play after the prepare time. See **IDirectMusicPerformance8::GetPrepareTime**.

DMUS_SEGF_GRID
: Play on a grid boundary.

DMUS_SEGF_BEAT

Play on a beat boundary.

DMUS_SEGF_MEASURE
Play on a measure boundary.

DMUS_SEGF_DEFAULT
Use the segment's default boundary.

DMUS_SEGF_NOINVALIDATE
Setting this flag in **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx** for a primary or control segment causes the new segment not to cause an invalidation. Without this flag, an invalidation occurs, cutting off and resetting any currently playing curve or note. This flag should be combined with DMUS_SEGF_AFTERPREPARETIME so that notes in the new segment do not play over notes played by the old segment.

DMUS_SEGF_ALIGN
The beginning of the segment can be aligned with a boundary, such as measure or beat, that has already passed. For this to happen, the segment must have a valid start point that falls before the next boundary. Start points can be defined in the segment, or one of the DMUS_SEGF_VALID_START_* flags can be used to define the granularity of valid start points. Any DMUS_SEGF_VALID_START_* flag takes effect only if a valid start point is not defined in the segment.

DMUS_SEGF_VALID_START_BEAT
Allow the start to occur on any beat. Used in combination with DMUS_SEGF_ALIGN.

DMUS_SEGF_VALID_START_GRID
Allow the start to occur on any grid. Used in combination with DMUS_SEGF_ALIGN.

DMUS_SEGF_VALID_START_TICK
Allow the start to occur at any time. Used in combination with DMUS_SEGF_ALIGN.

DMUS_SEGF_AUTOTRANSITION
Compose and play a transition segment, using the transition template.

DMUS_SEGF_AFTERQUEUETIME
Play after the queue time. This is the default for primary segments. Ignored if DMUS_SEGF_AFTERPREPARETIME is also set.

DMUS_SEGF_AFTERLATENCYTIME
Play after the latency time. This is true for all segments, so this flag currently has no effect.

DMUS_SEGF_SEGMENTEND
Play at the end of the primary segment that is playing at the start time. Any segments already queued after the currently playing primary segment are flushed. If no primary segment is playing, use other resolution flags.

DMUS_SEGF_MARKER
Play at next marker in the primary segment. If there are no markers, use other resolution flags.

DMUS_SEGF_TIMESIG_ALWAYS
Align start time with current time signature, even if there is no primary segment.

DMUS_SEGF_USE_AUDIOPATH
Use the audiopath embedded in the segment. Automatic downloading of bands must be enabled to ensure that the segment plays correctly.

DMUS_SEGF_VALID_START_MEASURE
Allow the start to occur at the beginning of a measure. Used in combination with DMUS_SEGF_ALIGN.

Remarks

The primary segment is the default control segment. The DMUS_SEGF_CONTROL flag can be used to make a secondary segment the control segment. If the DMUS_SEGF_CONTROL flag is set, DMUS_SEGF_SECONDARY is assumed. For more information, see Control Segments.

No more than one flag from each of the following groups should be specified.

Boundary
This flag controls the point in the currently playing primary segment at which the start point of the cued segment falls. It can be combined with DMUS_SEGF_MARKER, in which case the boundary flag will be used only if no marker exists in the primary segment.

DMUS_SEGF_BEAT
DMUS_SEGF_DEFAULT
DMUS_SEGF_GRID
DMUS_SEGF_MEASURE
DMUS_SEGF_QUEUE
DMUS_SEGF_SEGMENTEND

Alignment
This flag controls the segment start time of the cued segment, when its play time falls in the past. It must be combined with DMUS_SEGF_ALIGN.

DMUS_SEGF_VALID_START_BEAT
DMUS_SEGF_VALID_START_GRID
DMUS_SEGF_VALID_START_MEASURE
DMUS_SEGF_VALID_START_TICK

It is possible to combine one flag from each group. For example, combining DMUS_SEGF_MEASURE with DMUS_SEGF_ALIGN and DMUS_SEGF_VALID_START_BEAT causes the start point of the cued segment to fall at a measure boundary in the current primary segment. If this boundary has already passed, the cued segment starts playing at the next beat boundary within itself that is not aligned to a past time. For more information, see Segment Timing.

Requirements

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance8::Invalidate,
IDirectMusicPerformance8::PlaySegment,
IDirectMusicPerformance8::PlaySegmentEx, **IDirectMusicPerformance8::Stop**,
IDirectMusicSegment8::GetDefaultResolution,
IDirectMusicSegment8::SetDefaultResolution,
DMUS_TIME_RESOLVE_FLAGS

DMUS_SHAPET_TYPES

Used in the *wShape* parameter of the **IDirectMusicComposer8::ComposeSegmentFromShape** and **IDirectMusicComposer8::ComposeTemplateFromShape** methods to specify the desired pattern of the groove level.

```
typedef enum enumDMUS_SHAPET_TYPES {
    DMUS_SHAPET_FALLING  = 0,
    DMUS_SHAPET_LEVEL    = 1,
    DMUS_SHAPET_LOOPABLE = 2,
    DMUS_SHAPET_LOUD     = 3,
    DMUS_SHAPET_QUIET    = 4,
    DMUS_SHAPET_PEAKING  = 5,
    DMUS_SHAPET_RANDOM   = 6,
    DMUS_SHAPET_RISING   = 7,
    DMUS_SHAPET_SONG     = 8
} DMUS_SHAPET_TYPES;
```

Constants

DMUS_SHAPET_FALLING
: Groove level falls.

DMUS_SHAPET_LEVEL
: Groove level remains even.

DMUS_SHAPET_LOOPABLE
: Segment is arranged to loop back to the beginning.

DMUS_SHAPET_LOUD
: Groove level is high.

DMUS_SHAPET_QUIET
: Groove level is low.

DMUS_SHAPET_PEAKING
: Groove level rises to a peak, and then falls.

DMUS_SHAPET_RANDOM
Groove level is random.

DMUS_SHAPET_RISING
Groove level rises.

DMUS_SHAPET_SONG
Segment is in a song form. Several phrases of 6 to 8 bars are composed and put together to give a verse-chorus effect, with variations in groove level.

Requirements

Header: Declared in dmusici.h.

DMUS_STYLET_TYPES

Used in the **IDirectMusicPatternTrack8::SetPatternByName** and **IDirectMusicStyle8::EnumPattern** methods to specify a type of pattern.

```
typedef enum enumDMUS_STYLET_TYPES {
    DMUS_STYLET_PATTERN  = 0,
    DMUS_STYLET_MOTIF    = 1,
    DMUS_STYLET_FRAGMENT = 2,
} DMUS_STYLET_TYPES;
```

Constants

DMUS_STYLET_PATTERN
Normal pattern.

DMUS_STYLET_MOTIF
Motif pattern.

DMUS_STYLET_FRAGMENT
Not implemented.

Requirements

Header: Declared in dmusici.h.

DMUS_TIME_RESOLVE_FLAGS

Used in the **dwFlags** member of the **DMUS_PMSG** structure and in the *dwTimeResolveFlags* parameter of the **IDirectMusicPerformance8::GetResolvedTime** method.

```
typedef enum enumDMUS_TIME_RESOLVE_FLAGS {
    DMUS_TIME_RESOLVE_AFTERPREPARETIME =
```

```
        DMUS_SEGF_AFTERPREPARETIME,
    DMUS_TIME_RESOLVE_AFTERQUEUETIME   =
        DMUS_SEGF_AFTERQUEUETIME,
    DMUS_TIME_RESOLVE_AFTERLATENCYTIME =
        DMUS_SEGF_AFTERLATENCYTIME,
    DMUS_TIME_RESOLVE_GRID    = DMUS_SEGF_GRID,
    DMUS_TIME_RESOLVE_BEAT    = DMUS_SEGF_BEAT,
    DMUS_TIME_RESOLVE_MEASURE = DMUS_SEGF_MEASURE
    DMUS_TIME_RESOLVE_MARKER  = DMUS_SEGF_MARKER,
    DMUS_TIME_RESOLVE_SEGMENTEND =
  DMUS_SEGF_SEGMENTEND,
  } DMUS_TIME_RESOLVE_FLAGS;
```

Constants

DMUS_TIME_RESOLVE_AFTERPREPARETIME
Resolve to a time after the prepare time.

DMUS_TIME_RESOLVE_AFTERQUEUETIME
Resolve to a time after the queue time.

DMUS_TIME_RESOLVE_AFTERLATENCYTIME
Resolve to a time after the latency time.

DMUS_TIME_RESOLVE_GRID
Resolve to a time on a grid boundary.

DMUS_TIME_RESOLVE_BEAT
Resolve to a time on a beat boundary.

DMUS_TIME_RESOLVE_MEASURE
Resolve to a time on a measure boundary.

DMUS_TIME_RESOLVE_MARKER
Resolve to a marker.

DMUS_TIME_RESOLVE_SEGMENTEND
Resolve to the end of the segment.

Remarks

These flags can be used interchangeably with the corresponding **DMUS_SEGF_FLAGS**.

Requirements

Header: Declared in dmusici.h.

See Also

Timing

DMUS_TRACKF_FLAGS

Used in the *dwFlags* parameter of the **IDirectMusicTrack8::Play** and **IDirectMusicTrack8::PlayEx** methods.

```
typedef enum enumDMUS_TRACKF_FLAGS {
    DMUS_TRACKF_SEEK       = 1,
    DMUS_TRACKF_LOOP       = 2,
    DMUS_TRACKF_START      = 4,
    DMUS_TRACKF_FLUSH      = 8,
    DMUS_TRACKF_DIRTY      = 0x10,
    DMUS_TRACKF_NOTIFY_OFF = 0x20,
    DMUS_TRACKF_PLAYOFF    = 0x40,
    DMUS_TRACKF_LOOPEND    = 0x80,
    DMUS_TRACKF_STOP       = 0x100,
    DMUS_TRACKF_RECOMPOSE  = 0x200,
    DMUS_TRACKF_CLOCK      = 0x400,
} DMUS_TRACKF_FLAGS;
```

Constants

DMUS_TRACKF_SEEK
: **IDirectMusicTrack8::Play** was called in response to seeking, meaning that the *mtStart* parameter is not necessarily the same as the *mtEnd* of the previous call.

DMUS_TRACKF_LOOP
: **Play** was called in response to a loop.

DMUS_TRACKF_START
: This is the first call to **IDirectMusicTrack8::Play**. DMUS_TRACKF_SEEK can also be set if the track is not playing from the beginning.

DMUS_TRACKF_FLUSH
: **Play** was called in response to a flush or invalidation that requires the track to replay something that it played previously. In this case, DMUS_TRACKF_SEEK is set, as well.

DMUS_TRACKF_DIRTY
: A control segment has begun or ended. Tracks that normally wait until **mtNext** to call **IDirectMusicTrack8::GetParam** or **IDirectMusicTrack8::GetParamEx** should make the call right away, instead of waiting, because their data might now be invalid. For more information on setting control segments, see **DMUS_SEGF_FLAGS**.

DMUS_TRACKF_NOTIFY_OFF
: Track is not to send notifications.

DMUS_TRACKF_PLAYOFF
: Track is not to play, but can still send notifications.

DMUS_TRACKF_LOOPEND
: End of range is also a loop end.

DMUS_TRACKF_STOP
End of range is also end of playing this segment.

DMUS_TRACKF_RECOMPOSE
Track should be recomposed on each loop. A signpost track is recomposed using a different chord progression, and a melody formulation track is recomposed using different melody fragments.

DMUS_TRACKF_CLOCK
Time parameters are in reference time. Valid only for **IDirectMusicTrack8::PlayEx**.

Remarks

When **Play** is called in response to a repeat, DMUS_TRACKF_LOOP and DMUS_TRACKF_SEEK are set.

To support invalidation, tracks must support seeking..

Requirements

Header: Declared in dmplugin.h.

DMUS_VARIATIONT_TYPES

Used in the **DMUS_IO_PARTREF** structure to specify the way variations are selected in sequential commands.

```
typedef enum enumDMUS_VARIATIONT_TYPES
{
    DMUS_VARIATIONT_SEQUENTIAL      = 0,
    DMUS_VARIATIONT_RANDOM          = 1,
    DMUS_VARIATIONT_RANDOM_START    = 2,
    DMUS_VARIATIONT_NO_REPEAT       = 3,
    DMUS_VARIATIONT_RANDOM_ROW      = 4
} DMUS_VARIATIONT_TYPES;
```

Constants

DMUS_VARIATIONT_SEQUENTIAL
Play matching variations sequentially, in the order loaded, starting with the first.

DMUS_VARIATIONT_RANDOM
Select a random matching variation. This is the behavior in versions prior to DirectX 8.0.

DMUS_VARIATIONT_RANDOM_START
Play matching variations sequentially, in the order loaded, starting at a random point in the sequence.

DMUS_VARIATIONT_NO_REPEAT

Play randomly, but do not play the same variation twice.

DMUS_VARIATIONT_RANDOM_ROW
Play randomly, but do not repeat any variation until all have played.

Requirements

Header: Declared in dmusicf.h.

See Also

DMUS_COMMAND_PARAM, **DMUS_COMMAND_PARAM_2**, **DMUS_IO_COMMAND**, **DMUS_PATTERNT_TYPES**

DirectMusic Return Values

This section provides a brief explanation of the various error codes that can be returned by DirectMusic methods. For a list of the specific codes that each method can return, see the individual method descriptions. The lists given there are not necessarily comprehensive.

Error codes are presented in the following sections:

- DirectMusic Return Values By Number
- DirectMusic Return Values By Name

DirectMusic Return Values By Number

The following table lists DirectMusic return values sorted by hexadecimal value. For a description, click on the constant.

Hexadecimal	Constant
0x00000000	S_OK
0x00000001	S_FALSE
0x08781091	DMUS_S_PARTIALLOAD
0x08781092	DMUS_S_PARTIALDOWNLOAD
0x08781200	DMUS_S_REQUEUE
0x08781201	DMUS_S_FREE
0x08781202	DMUS_S_END
0x08781210	DMUS_S_STRING_TRUNCATED
0x08781211	DMUS_S_LAST_TOOL
0x08781212	DMUS_S_OVER_CHORD
0x08781213	DMUS_S_UP_OCTAVE

0x08781214	DMUS_S_DOWN_OCTAVE
0x08781215	DMUS_S_NOBUFFERCONTROL
0x08781216	DMUS_S_GARBAGE_COLLECTED
0x80004001	E_NOTIMPL
0x80004002	E_NOINTERFACE
0x80004003	E_POINTER
0x80040110	CLASS_E_NOAGGREGATION
0x80040110	CLASS_E_NOAGGREGATION
0x80040154	REGDB_E_CLASSNOTREG
0x8007000E	E_OUTOFMEMORY
0x80070057	E_FAIL
0x80070057	E_INVALIDARG
0x88781101	DMUS_E_DRIVER_FAILED
0x88781102	DMUS_E_PORTS_OPEN
0x88781103	DMUS_E_DEVICE_IN_USE
0x88781104	DMUS_E_INSUFFICIENTBUFFER
0x88781105	DMUS_E_BUFFERNOTSET
0x88781106	DMUS_E_BUFFERNOTAVAILABLE
0x88781108	DMUS_E_NOTADLSCOL
0x88781109	DMUS_E_INVALIDOFFSET
0x88781111	DMUS_E_ALREADY_LOADED
0x88781113	DMUS_E_INVALIDPOS
0x88781114	DMUS_E_INVALIDPATCH
0x88781115	DMUS_E_CANNOTSEEK
0x88781116	DMUS_E_CANNOTWRITE
0x88781117	DMUS_E_CHUNKNOTFOUND
0x88781119	DMUS_E_INVALID_DOWNLOADID
0x88781120	DMUS_E_NOT_DOWNLOADED_TO_PORT
0x88781121	DMUS_E_ALREADY_DOWNLOADED
0x88781122	DMUS_E_UNKNOWNDOWNLOAD
0x88781123	DMUS_E_SET_UNSUPPORTED
0x88781124	DMUS_E_GET_UNSUPPORTED
0x88781125	DMUS_E_NOTMONO
0x88781126	DMUS_E_BADARTICULATION
0x88781127	DMUS_E_BADINSTRUMENT
0x88781128	DMUS_E_BADWAVELINK
0x88781128	DMUS_E_BUFFER_EMPTY
0x8878112A	DMUS_E_NOTPCM

0x8878112B	DMUS_E_BADWAVE
0x8878112C	DMUS_E_BADOFFSETTABLE
0x8878112D	DMUS_E_UNKNOWN_PROPERTY
0x8878112E	DMUS_E_NOSYNTHSINK
0x8878112F	DMUS_E_ALREADYOPEN
0x88781130	DMUS_E_ALREADYCLOSED
0x88781131	DMUS_E_SYNTHNOTCONFIGURED
0x88781132	DMUS_E_SYNTHACTIVE
0x88781133	DMUS_E_CANNOTREAD
0x88781134	DMUS_E_DMUSIC_RELEASED
0x88781136	DMUS_E_BUFFER_FULL
0x88781137	DMUS_E_PORT_NOT_CAPTURE
0x88781138	DMUS_E_PORT_NOT_RENDER
0x88781139	DMUS_E_DSOUND_NOT_SET
0x8878113A	DMUS_E_ALREADY_ACTIVATED
0x8878113B	DMUS_E_INVALIDBUFFER
0x8878113C	DMUS_E_WAVEFORMATNOTSUPPORTED
0x8878113D	DMUS_E_SYNTHINACTIVE
0x8878113E	DMUS_E_DSOUND_ALREADY_SET
0x8878113F	DMUS_E_INVALID_EVENT
0x88781150	DMUS_E_UNSUPPORTED_STREAM
0x88781151	DMUS_E_ALREADY_INITED
0x88781152	DMUS_E_INVALID_BAND
0x88781155	DMUS_E_TRACK_HDR_NOT_FIRST_CK
0x88781156	DMUS_E_TOOL_HDR_NOT_FIRST_CK
0x88781157	DMUS_E_INVALID_TRACK_HDR
0x88781158	DMUS_E_INVALID_TOOL_HDR
0x88781159	DMUS_E_ALL_TOOLS_FAILED
0x88781160	DMUS_E_ALL_TRACKS_FAILED
0x88781161	DMUS_E_NOT_FOUND
0x88781162	DMUS_E_NOT_INIT
0x88781163	DMUS_E_TYPE_DISABLED
0x88781164	DMUS_E_TYPE_UNSUPPORTED
0x88781165	DMUS_E_TIME_PAST
0x88781166	DMUS_E_TRACK_NOT_FOUND
0x88781167	DMUS_E_TRACK_NO_CLOCKTIME_SUPPORT
0x88781170	DMUS_E_NO_MASTER_CLOCK
0x88781170	DMUS_E_NOARTICULATION

0x88781180	DMUS_E_LOADER_NOCLASSID
0x88781181	DMUS_E_LOADER_BADPATH
0x88781182	DMUS_E_LOADER_FAILEDOPEN
0x88781183	DMUS_E_LOADER_FORMATNOTSUPPORTED
0x88781184	DMUS_E_LOADER_FAILEDCREATE
0x88781185	DMUS_E_LOADER_OBJECTNOTFOUND
0x88781186	DMUS_E_LOADER_NOFILENAME
0x88781200	DMUS_E_INVALIDFILE
0x88781201	DMUS_E_ALREADY_EXISTS
0x88781202	DMUS_E_OUT_OF_RANGE
0x88781203	DMUS_E_SEGMENT_INIT_FAILED
0x88781204	DMUS_E_ALREADY_SENT
0x88781205	DMUS_E_CANNOT_FREE
0x88781206	DMUS_E_CANNOT_OPEN_PORT
0x88781207	DMUS_E_CANNOT_CONVERT
0x88781210	DMUS_E_DESCEND_CHUNK_FAIL
0x88781211	DMUS_E_NOT_LOADED
0x88781213	DMUS_E_SCRIPT_LANGUAGE_INCOMPATIBLE
0x88781214	DMUS_E_SCRIPT_UNSUPPORTED_VARTYPE
0x88781215	DMUS_E_SCRIPT_ERROR_IN_SCRIPT
0x88781216	DMUS_E_SCRIPT_CANTLOAD_OLEAUT32
0x88781217	DMUS_E_SCRIPT_LOADSCRIPT_ERROR
0x88781218	DMUS_E_SCRIPT_INVALID_FILE
0x88781219	DMUS_E_INVALID_SCRIPTTRACK
0x8878121A	DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
0x8878121B	DMUS_E_SCRIPT_ROUTINE_NOT_FOUND
0x8878121C	DMUS_E_SCRIPT_CONTENT_READONLY
0x8878121D	DMUS_E_SCRIPT_NOT_A_REFERENCE
0x8878121E	DMUS_E_SCRIPT_VALUE_NOT_SUPPORTED
0x88781220	DMUS_E_INVALID_SEGMENTTRIGGERTRACK
0x88781221	DMUS_E_INVALID_LYRICSTRACK
0x88781222	DMUS_E_INVALID_PARAMCONTROLTRACK
0x88781223	DMUS_E_AUDIOVBSCRIPT_SYNTAXERROR
0x88781224	DMUS_E_AUDIOVBSCRIPT_RUNTIMEERROR
0x88781225	DMUS_E_AUDIOVBSCRIPT_OPERATIONFAILURE
0x88781226	DMUS_E_AUDIOPATHS_NOT_VALID
0x88781227	DMUS_E_AUDIOPATHS_IN_USE

0x88781228	DMUS_E_NO_AUDIOPATH_CONFIG
0x88781229	DMUS_E_AUDIOPATH_INACTIVE
0x8878122A	DMUS_E_AUDIOPATH_NOBUFFER
0x8878122B	DMUS_E_AUDIOPATH_NOPORT
0x8878122C	DMUS_E_NO_AUDIOPATH
0x8878122D	DMUS_E_INVALIDCHUNK
0x8878122E	DMUS_E_AUDIOPATH_NOGLOBALFXBUFFER
0x8878122F	DMUS_E_INVALID_CONTAINER_OBJECT

DirectMusic Return Values by Name

The following list describes all DirectMusic return values. To find a constant from its value, see DirectMusic Return Values By Number.

CLASS_E_NOAGGREGATION
: Aggregation is not supported. The LPUNKNOWN parameter should be set to NULL.

DMUS_E_ALL_TOOLS_FAILED
: The graph object was unable to load all tools from the **IStream** object data, perhaps because of errors in the stream or because the tools are incorrectly registered on the client.

DMUS_E_ALL_TRACKS_FAILED
: The segment object was unable to load all tracks from the **IStream** object data, perhaps because of errors in the stream or because the tracks are incorrectly registered on the client.

DMUS_E_ALREADY_ACTIVATED
: The port has been activated, and the parameter cannot be changed.

DMUS_E_ALREADY_DOWNLOADED
: The buffer has already been downloaded.

DMUS_E_ALREADY_EXISTS
: The tool is already contained in the graph. You must create a new instance.

DMUS_E_ALREADY_INITED
: The object has already been initialized.

DMUS_E_ALREADY_LOADED
: A DLS collection is already open.

DMUS_E_ALREADY_SENT
: The message has already been sent.

DMUS_E_ALREADYCLOSED
: The port is not open.

DMUS_E_ALREADYOPEN
: The port is already open.

DMUS_E_AUDIOPATH_INACTIVE

The audiopath is inactive, perhaps because the performance has been closed down.

DMUS_E_AUDIOPATH_NOBUFFER
The audiopath could not be created because a requested buffer could not be created.

DMUS_E_AUDIOPATH_NOGLOBALFXBUFFER
An attempt was made to create an audiopath that sends to a nonexistent global effects buffer.

DMUS_E_AUDIOPATH_NOPORT
The audiopath could not be used for playback because it lacked port assignments.

DMUS_E_AUDIOPATHS_IN_USE
The performance has set up audiopaths, so performance channels cannot be allocated.

DMUS_E_AUDIOPATHS_NOT_VALID
Performance channels have been set up by using **IDirectMusicPerformance8::AssignPChannel**, so the performance cannot support audiopaths.

DMUS_E_AUDIOVBSCRIPT_OPERATIONFAILURE
A script routine written in AudioVBScript failed because a function outside the script failed to complete.

DMUS_E_AUDIOVBSCRIPT_RUNTIMEERROR
A script routine written in AudioVBScript failed because an invalid operation occurred—for example, adding an integer to an object, or attempting to call a routine that does not exist.

DMUS_E_AUDIOVBSCRIPT_SYNTAXERROR
A script routine written in AudioVBScript could not be read because it contained a statement not allowed by the language.

DMUS_E_BADARTICULATION
Invalid articulation chunk in DLS collection.

DMUS_E_BADINSTRUMENT
Invalid instrument chunk in DLS collection.

DMUS_E_BADOFFSETTABLE
The offset table has errors.

DMUS_E_BADWAVE
Corrupt wave header.

DMUS_E_BADWAVELINK
The wave-link chunk in DLS collection points to invalid wave.

DMUS_E_BUFFER_EMPTY
There is no data in the buffer.

DMUS_E_BUFFER_FULL
The specified number of bytes exceeds the maximum buffer size.

DMUS_E_BUFFERNOTAVAILABLE
The buffer is not available for download.

DMUS_E_BUFFERNOTSET
No buffer was prepared for the data.

DMUS_E_CANNOT_CONVERT
The requested conversion between music and MIDI values could not be made. This usually occurs when the provided **DMUS_CHORD_KEY** structure has an invalid chord or scale pattern.

DMUS_E_CANNOT_FREE
The message could not be freed, either because it was not allocated or because it has already been freed.

DMUS_E_CANNOT_OPEN_PORT
The default system port could not be opened.

DMUS_E_CANNOTREAD
An error occurred when trying to read from the **IStream** object.

DMUS_E_CANNOTSEEK
The **IStream** object does not support **Seek**.

DMUS_E_CANNOTWRITE
The **IStream** object does not support **Write**.

DMUS_E_CHUNKNOTFOUND
A chunk with the specified header could not be found.

DMUS_E_DESCEND_CHUNK_FAIL
An attempt to descend into a chunk failed.

DMUS_E_DEVICE_IN_USE
The device is in use, possibly by a non-DirectMusic client, and cannot be opened again.

DMUS_E_DMUSIC_RELEASED
The operation cannot be performed because the final instance of the DirectMusic object was released. Ports cannot be used after final release of the DirectMusic object.

DMUS_E_DRIVER_FAILED
An unexpected error was returned from a device driver, indicating possible failure of the driver or hardware.

DMUS_E_DSOUND_ALREADY_SET
A DirectSound object has already been set.

DMUS_E_DSOUND_NOT_SET
The port could not be created because no DirectSound object has been specified.

DMUS_E_GET_UNSUPPORTED
Getting the parameter is not supported.

DMUS_E_INSUFFICIENTBUFFER
The buffer is not large enough for the requested operation.

DMUS_E_INVALID_BAND
The file does not contain a valid band.

DMUS_E_INVALID_CONTAINER_OBJECT
The file does not contain a valid container object.

DMUS_E_INVALID_DOWNLOADID
An invalid download identifier was used in the process of creating a download buffer.

DMUS_E_INVALID_EVENT
The event either is not a valid MIDI message or makes use of running status and cannot be packed into the buffer.

DMUS_E_INVALID_LYRICSTRACK
The file contains an invalid lyrics track.

DMUS_E_INVALID_PARAMCONTROLTRACK
The file contains an invalid parameter control track.

DMUS_E_INVALID_SCRIPTTRACK
The file contains an invalid script track.

DMUS_E_INVALID_SEGMENTTRIGGERTRACK
The file contains an invalid segment trigger track.

DMUS_E_INVALID_TOOL_HDR
The **IStream** object's data contains an invalid tool header and cannot be read by the graph object.

DMUS_E_INVALID_TRACK_HDR
The **IStream** object's data contains an invalid track header and cannot be read by the segment object.

DMUS_E_INVALIDBUFFER
An invalid DirectSound buffer was handed to a port.

DMUS_E_INVALIDCHUNK
Invalid data was found in a RIFF file chunk.

DMUS_E_INVALIDFILE
Not a valid file.

DMUS_E_INVALIDOFFSET
Wave chunks in the DLS collection file are at incorrect offsets.

DMUS_E_INVALIDPATCH
No instrument in the collection matches the patch number.

DMUS_E_INVALIDPOS
Error reading wave data from a DLS collection. Indicates bad file.

DMUS_E_LOADER_BADPATH
The file path is invalid.

DMUS_E_LOADER_FAILEDCREATE
The object could not be found or created.

DMUS_E_LOADER_FAILEDOPEN
File open failed because the file does not exist or is locked.

DMUS_E_LOADER_FORMATNOTSUPPORTED
The object cannot be loaded because the data format is not supported.

DMUS_E_LOADER_NOCLASSID
No class identifier was supplied in the object description.

DMUS_E_LOADER_NOFILENAME

No file name was supplied in the object description.

DMUS_E_LOADER_OBJECTNOTFOUND
The object was not found.

DMUS_E_NO_AUDIOPATH
An attempt was made to play on a nonexistent audiopath.

DMUS_E_NO_AUDIOPATH_CONFIG
The object does not contain an embedded audiopath configuration.

DMUS_E_NO_MASTER_CLOCK
There is no master clock in the performance. Be sure to call the **IDirectMusicPerformance8::Init** method.

DMUS_E_NOARTICULATION
Articulation missing from an instrument in the DLS collection.

DMUS_E_NOSYNTHSINK
No sink is connected to the synthesizer.

DMUS_E_NOT_DOWNLOADED_TO_PORT
The object cannot be unloaded because it is not present on the port.

DMUS_E_NOT_FOUND
The requested item is not contained by the object.

DMUS_E_NOT_INIT
A required object is not initialized or failed to initialize.

DMUS_E_NOT_LOADED
An attempt to use this object failed because it was not loaded.

DMUS_E_NOTADLSCOL
The object being loaded is not a valid DLS collection.

DMUS_E_NOTMONO
The wave chunk has more than one interleaved channel. DLS format requires mono.

DMUS_E_NOTPCM
Wave data is not in PCM format.

DMUS_E_OUT_OF_RANGE
The requested time is outside the range of the segment.

DMUS_E_PORT_NOT_CAPTURE
The port is not a capture port.

DMUS_E_PORT_NOT_RENDER
The port is not an output port.

DMUS_E_PORTS_OPEN
The requested operation cannot be performed while there are instantiated ports in any process in the system.

DMUS_E_SCRIPT_CANTLOAD_OLEAUT32
Loading of Oleaut32.dll failed. ActiveX scripting languages require use of oleaut32.dll. On platforms where this file is not present, only the AudioVBScript language can be used.

DMUS_E_SCRIPT_CONTENT_READONLY

Script variables for content referenced or embedded in a script cannot be set.

DMUS_E_SCRIPT_ERROR_IN_SCRIPT
An error was encountered while parsing or executing the script.

DMUS_E_SCRIPT_INVALID_FILE
The script file is invalid.

DMUS_E_SCRIPT_LANGUAGE_INCOMPATIBLE
The ActiveX scripting engine for the script's language is not compatible with DirectMusic.

DMUS_E_SCRIPT_LOADSCRIPT_ERROR
The script that was loaded contains an error.

DMUS_E_SCRIPT_NOT_A_REFERENCE
An attempt was made to set a script's variable by reference to a value that is not an object type.

DMUS_E_SCRIPT_ROUTINE_NOT_FOUND
The script does not contain a routine with the specified name.

DMUS_E_SCRIPT_UNSUPPORTED_VARTYPE
A variant was used that had a type not supported by DirectMusic.

DMUS_E_SCRIPT_VALUE_NOT_SUPPORTED
An attempt was made to set a script's variable by value to an object that does not support a default value property.

DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
The script does not contain a variable with the specified name.

DMUS_E_SEGMENT_INIT_FAILED
Segment initialization failed, probably because of a critical memory situation.

DMUS_E_SET_UNSUPPORTED
Setting the parameter is not supported.

DMUS_E_SYNTHACTIVE
The synthesizer has been activated, and the parameter cannot be changed.

DMUS_E_SYNTHINACTIVE
The synthesizer has not been activated and cannot process data.

DMUS_E_SYNTHNOTCONFIGURED
The synthesizer is not properly configured or opened.

DMUS_E_TIME_PAST
The time requested is in the past.

DMUS_E_TOOL_HDR_NOT_FIRST_CK
The stream object's data does not have a tool header as the first chunk and, therefore, cannot be read by the graph object.

DMUS_E_TRACK_HDR_NOT_FIRST_CK
The stream object's data does not have a track header as the first chunk and, therefore, cannot be read by the segment object.

DMUS_E_TRACK_NO_CLOCKTIME_SUPPORT
The track does not support clock-time playback or parameter retrieval.

DMUS_E_TRACK_NOT_FOUND

There is no track of the requested type.

DMUS_E_TYPE_DISABLED
A track parameter is unavailable because it has been disabled.

DMUS_E_TYPE_UNSUPPORTED
Parameter is unsupported on this track.

DMUS_E_UNKNOWN_PROPERTY
The property set or item is not implemented by this port.

DMUS_E_UNKNOWNDOWNLOAD
The synthesizer does not support this type of download.

DMUS_E_UNSUPPORTED_STREAM
The **IStream** object does not contain data supported by the loading object.

DMUS_E_WAVEFORMATNOTSUPPORTED
Invalid buffer format was handed to the synthesizer sink.

DMUS_S_DOWN_OCTAVE
The note has been lowered by one or more octaves to fit within the range of MIDI values.

DMUS_S_END
The operation succeeded and reached the end of the data.

DMUS_S_FREE
The allocated memory should be freed.

DMUS_S_GARBAGE_COLLECTED
The requested operation was not performed because the object has been released.

DMUS_S_LAST_TOOL
There are no more tools in the graph.

DMUS_S_NOBUFFERCONTROL
Although the audio output from the port is routed to the same device as the given DirectSound buffer, buffer controls such as pan and volume do not affect the output.

DMUS_S_OVER_CHORD
No MIDI values have been calculated because the music value has the note at a position higher than the top note of the chord.

DMUS_S_PARTIALDOWNLOAD
Some instruments could not be downloaded to the port.

DMUS_S_PARTIALLOAD
The object could only load partially. This can happen if some components, such as embedded tracks and tools, are not registered properly. It can also happen if some content is missing; for example, if a segment uses a DLS collection that is not in the loader's current search directory.

DMUS_S_REQUEUE
The message should be passed to the next tool.

DMUS_S_STRING_TRUNCATED
The method succeeded, but the returned string was truncated.

DMUS_S_UP_OCTAVE

The note has been raised by one or more octaves to fit within the range of MIDI values.

E_FAIL
The method did not succeed.

E_INVALIDARG
Invalid argument. Often, this error results from failing to initialize the **dwSize** member of a structure before passing it to the method.

E_NOINTERFACE
No object interface is available.

E_NOTIMPL
The method is not implemented. This value might be returned if a driver does not support a feature necessary for the operation.

E_OUTOFMEMORY
Insufficient memory to complete the task.

E_POINTER
An invalid pointer, usually NULL, was passed as a parameter.

REGDB_E_CLASSNOTREG
The object class is not registered.

S_FALSE
The method succeeded, but there was nothing to do.

S_OK
The operation was completed successfully.

DirectSound C/C++ Reference

This section contains reference information for the Microsoft® DirectSound® application programming interface (API). This section contains reference information for the API elements that DirectSound provides. Reference material is divided into the following categories.

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

DirectSound Interfaces

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

- **IDirectSound8**
- **IDirectSound3DBuffer8**
- **IDirectSound3DListener8**
- **IDirectSoundBuffer8**
- **IDirectSoundCapture8**
- **IDirectSoundCaptureBuffer8**
- **IDirectSoundCaptureFXAec8**
- **IDirectSoundCaptureFXNoiseSuppress8**
- **IDirectSoundFullDuplex8**
- **IDirectSoundFXChorus8**
- **IDirectSoundFXCompressor8**
- **IDirectSoundFXDistortion8**
- **IDirectSoundFXEcho8**
- **IDirectSoundFXFlanger8**
- **IDirectSoundFXGargle8**
- **IDirectSoundFXI3DL2Reverb8**
- **IDirectSoundFXParamEq8**
- **IDirectSoundNotify8**
- **IKsPropertySet**

For DirectX 8.0, only **IDirectSound8, IDirectSoundBuffer8**, and **IDirectSoundCaptureBuffer8** are new interfaces that supersede interfaces with similar, unnumbered names. Other interface names ending in **8** are simple defines for unchanged or new unnumbered interfaces. For example, **IDirectSoundCapture8** is exactly equivalent to **IDirectSoundCapture**.

All interfaces that have a define ending in **8** are documented under that defined name, and it is recommended that you always use this name in your code. When recompiling for future versions, you can substitute the latest number in all your declarations and be sure of getting the latest version of any updated interfaces.

When a method takes an interface pointer as a parameter, you can pass in the newer version even where the method is declared as accepting the older version. For example, a pointer to either **IDirectSoundBuffer** or **IDirectSoundBuffer8** can be passed to **IDirectSound8::DuplicateSoundBuffer** as the *pDSBufferOriginal* parameter.

When a method returns an interface, however, it is usually the older interface, and the newer interface must be obtained by using **QueryInterface**. This is the case, for example, with the *ppDSBuffer* parameter of **IDirectSound8::CreateSoundBuffer**.

IDirectSound8

Used to create buffer objects and set up the environment.

The **IDirectSound8** interface supersedes **IDirectSound** and adds new methods. See Remarks.

Obtain this interface by using the **DirectSoundCreate8** or **DirectSoundFullDuplexCreate8** function, or by using **CoCreateInstance** or **CoCreateInstanceEx** to create an object of class CLSID_DirectSound8.

The methods of the **IDirectSound8** interface can be organized into the following groups:

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

The **IDirectSound8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTSOUND8** type is defined as a pointer to the **IDirectSound8** interface:

```
typedef struct IDirectSound8    *LPDIRECTSOUND8;
```

Remarks

Only objects of class CLSID_DirectSound8 support this interface. All DirectSound objects created by **DirectSoundCreate8** and **DirectSoundFullDuplexCreate8** fall into this category. Objects of class CLSID_DirectSound, which include all those created by using **DirectSoundCreate**, support only the earlier **IDirectSound** interface.

The behavior of CLSID_DirectSound8 objects is somewhat different than that of CLSID_DirectSound objects. These differences are found in the **IDirectSound**

interface as well as the **IDirectSound8** interface. Specific differences in the behavior of the newer object include the following:

- In **IDirectSound8::CreateSoundBuffer**, DSBCAPS_CTRL3D cannot be set on a buffer with a non-mono format, and DSBCAPS_CTRL3D cannot be combined with DSBCAPS_CTRLPAN.
- New buffer creation flags are supported.
- Buffers are not filled with silence on creation.
- The **IDirectSoundBuffer** interface returned by **CreateSoundBuffer** can be queried for the **IDirectSoundBuffer8** interface.
- Wave formats in the **DSBUFFERDESC** structure that have the WAVE_FORMAT_EXTENSIBLE format tag are checked more strictly for validity.

Requirements

Header: Declared in dsound.h.

IDirectSound8::Compact

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

```
HRESULT Compact();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED
DSERR_UNINITIALIZED

Remarks

The application's DirectSound object must have at least the DSSCL_PRIORITY cooperative level. See **IDirectSound8::SetCooperativeLevel**.

This method will fail if any operations are in progress.

Requirements

Header: Declared in dsound.h

IDirectSound8::CreateSoundBuffer

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

```
HRESULT CreateSoundBuffer(
  LPCDSBUFFERDESC pcDSBufferDesc,
  LPDIRECTSOUNDBUFFER * ppDSBuffer,
  LPUNKNOWN pUnkOuter
);
```

Parameters

pcDSBufferDesc
Address of a **DSBUFFERDESC** structure that describes the sound buffer to create.

ppDSBuffer
Address of a variable that receives the **IDirectSoundBuffer** interface of the new buffer object. Use **QueryInterface** to obtain **IDirectSoundBuffer8**. **IDirectSoundBuffer8** is not available for primary buffers. See Remarks.

pUnkOuter
Address of the controlling object's **IUnknown** interface for COM aggregation. Must be NULL.

Return Values

If the method succeeds, the return value is DS_OK, or DS_NO_VIRTUALIZATION if a requested 3-D algorithm was not available and stereo panning was substituted. See the description of the **guid3DAlgorithm** member of **DSBUFFERDESC**.

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

DSERR_ALLOCATED
DSERR_CONTROLUNAVAIL
DSERR_BADFORMAT
DSERR_BUFFERTOOSMALL
DSERR_DS8_REQUIRED
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_NOAGGREGATION
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

Remarks

DirectSound does not initialize the contents of the buffer, and the application cannot assume that it contains silence.

If the application is multithreaded, the thread that creates the buffer must continue to exist for the lifetime of the buffer. Buffers created on WDM drivers stop playing when the thread is terminated.

If an attempt is made to create a buffer with the DSBCAPS_LOCHARDWARE flag on a system where hardware acceleration is not available, the method fails with either DSERR_CONTROLUNAVAIL or DSERR_INVALIDCALL, depending on the operating system.

Requirements

Header: Declared in dsound.h.

See Also

Creating Secondary Buffers, **IDirectSound8::DuplicateSoundBuffer**

IDirectSound8::DuplicateSoundBuffer

Creates a new DirectSoundBuffer object that duplicates or shares the original buffer's memory.

```
HRESULT DuplicateSoundBuffer(
  LPDIRECTSOUNDBUFFER pDSBufferOriginal,
  LPDIRECTSOUNDBUFFER * ppDSBufferDuplicate
);
```

Parameters

pDSBufferOriginal
Address of the **IDirectSoundBuffer** or **IDirectSoundBuffer8** interface of the buffer to duplicate.

ppDSBufferDuplicate
Address of a variable that receives the **IDirectSoundBuffer** interface pointer for the new buffer. The **IDirectSoundBuffer8** interface cannot be obtained from duplicate buffers.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_ALLOCATED
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

Remarks

This method is not valid for buffers created with the DSBCAPS_CTRLFX flag.

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

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

Although duplicate buffers often share the same memory, this is not always the case. Changes in the data stored in one buffer are not necessarily reflected in its duplicates.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound8::CreateSoundBuffer

IDirectSound8::GetCaps

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

```
HRESULT GetCaps(
  LPDSCAPS pDSCaps
);
```

Parameters

pDSCaps
Address of a **DSCAPS** structure that receives the capabilities of this sound device.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_GENERIC

DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

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

Because of resource-sharing requirements, the maximum capabilities in one area might be available only at the cost of another area.

Requirements

Header: Declared in dsound.h.

IDirectSound8::GetSpeakerConfig

Retrieves the speaker configuration.

```
HRESULT GetSpeakerConfig(
  LPDWORD pdwSpeakerConfig
);
```

Parameters

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

DSSPEAKER_5POINT1
The audio is played through a speaker arrangement of surround speakers with a subwoofer.

DSSPEAKER_HEADPHONE
The audio is played through headphones.

DSSPEAKER_MONO
The audio is played through a single speaker.

DSSPEAKER_QUAD
The audio is played through quadraphonic speakers.

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

DSSPEAKER_SURROUND
The audio is played through surround speakers.

DSSPEAKER_STEREO can be combined with one of the following values:

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

DSSPEAKER_GEOMETRY_NARROW
The speakers are directed over an arc of 10 degrees.
DSSPEAKER_GEOMETRY_MIN
The speakers are directed over an arc of 5 degrees.
DSSPEAKER_GEOMETRY_MAX
The speakers are directed over an arc of 180 degrees.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

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

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound8::SetSpeakerConfig, Speaker Configuration

IDirectSound8::Initialize

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

```
HRESULT Initialize(
  LPCGUID pcGuidDevice
);
```

Parameters

pcGuidDevice

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

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_ALREADYINITIALIZED

DSERR_GENERIC

DSERR_INVALIDPARAM

DSERR_NODRIVER

Remarks

This method is provided for compliance with the Component Object Model (COM) protocol. If the **DirectSoundCreate8** function was used to create the DirectSound object, this method returns DSERR_ALREADYINITIALIZED. If **IDirectSound8::Initialize** is not called when using **CoCreateInstance** to create the DirectSound object, any method called afterward returns DSERR_UNINITIALIZED.

Requirements

Header: Declared in dsound.h.

IDirectSound8::SetCooperativeLevel

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

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

Parameters

hwnd

Handle to the application window.

dwLevel

Requested priority level. Specify one of the following values:

DSSCL_EXCLUSIVE

For DirectX 8.0 and later, has the same effect as DSSCL_PRIORITY. For previous versions, sets the application to the exclusive level. This means that when it has the input focus, the application will be the only one audible; sounds from applications with the DSBCAPS_GLOBALFOCUS flag set will be muted. With this level, it also has all the privileges of the DSSCL_PRIORITY level. DirectSound will restore the hardware format, as specified by the most recent call to the **IDirectSoundBuffer8::SetFormat** method, after the application gains the input focus.

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

DSSCL_PRIORITY
Sets the application to the priority level. Applications with this cooperative level can call the **IDirectSoundBuffer8::SetFormat** and **IDirectSound8::Compact** methods.

DSSCL_WRITEPRIMARY
This is the highest priority level. The application has write access to the primary sound buffers. No secondary sound buffers can be played. This level cannot be set if the DirectSound driver is being emulated for the device; that is, if the **IDirectSound8::GetCaps** method returns the DSCAPS_EMULDRIVER flag in the **DSCAPS** structure.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_ALLOCATED
DSERR_INVALIDPARAM
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

Remarks

The application must set the cooperative level by calling this method before its buffers can be played. The recommended cooperative level is DSSCL_PRIORITY. For more information, see Cooperative Levels.

Do not call this method if any buffers are locked.

Requirements

Header: Declared in dsound.h.

IDirectSound8::SetSpeakerConfig

Specifies the speaker configuration of the DirectSound object.

It is not recommended that applications use this method. See Speaker Configuration.

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

Parameters

dwSpeakerConfig

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

DSSPEAKER_5POINT1
The audio is played through a speaker arrangement of surround speakers with a subwoofer.

DSSPEAKER_HEADPHONE
The speakers are headphones.

DSSPEAKER_MONO
The speakers are monaural.

DSSPEAKER_QUAD
The speakers are quadraphonic.

DSSPEAKER_STEREO
The speakers are stereo (default value).

DSSPEAKER_SURROUND
The speakers are surround sound.

DSSPEAKER_STEREO can be combined with one of the following values:

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

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

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

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

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM

DSERR_UNINITIALIZED

Remarks

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

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound8::GetSpeakerConfig

IDirectSound8::VerifyCertification

Ascertains whether the DirectSound device is certified for DirectX.

```
VerifyCertification(
  LPDWORD pdwCertified
);
```

Parameters

pdwCertified

Address of a **DWORD** variable that receives one of the following values.

DS_CERTIFIED
: Driver is certified for DirectSound.

DS_UNCERTIFIED
: Driver is not certified for DirectSound.

Return Values

If the function succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM

DSERR_UNSUPPORTED

Remarks

On emulated devices, the method returns DSERR_UNSUPPORTED. Emulated devices are identified by the DSCAPS_EMULDRIVER flag in the **dwFlags** member of **DSCAPS**.

Requirements

Header: Declared in dsound.h.

IDirectSound3DBuffer8

Used to retrieve and set parameters that describe the position, orientation, and environment of a sound buffer in 3-D space.

IDirectSound3DBuffer8 is a define for **IDirectSoundBuffer**. The two interface names are interchangeable.

The **IDirectSound3DBuffer8** interface is obtained from an **IDirectSoundBuffer8** interface by using the **IDirectSoundBuffer8::QueryInterface** method. It can also be retrieved from an audio path.

The methods of the **IDirectSound3DBuffer8** interface can be organized into the following groups:

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

The **IDirectSound3DBuffer8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTSOUND3DBUFFER** type is defined as a pointer to the **IDirectSound3DBuffer8** interface:

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

Requirements

Header: Declared in dsound.h.

See Also

Obtaining the 3-D Buffer Object, Retrieving Objects from an Audiopath

IDirectSound3DBuffer8::GetAllParameters

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

```
HRESULT GetAllParameters(
  LPDS3DBUFFER pDs3dBuffer
);
```

Parameters

pDs3dBuffer
Address of a **DS3DBUFFER** structure that receives the information describing the 3-D characteristics of the sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSound3DBuffer8::GetConeAngles

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

```
HRESULT GetConeAngles(
  LPDWORD pdwInsideConeAngle,
  LPDWORD pdwOutsideConeAngle
);
```

Parameters

pdwInsideConeAngle
Addresses of a variable that receives the inside angle of the sound projection cone, in degrees. This is the angle within which the sound is at its normal volume.

pdwOutsideConeAngle
Addresses of a variable that receives the outside angle of the sound projection cone, in degrees. This is the angle outside of which the sound is at its outside volume.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::GetConeOutsideVolume, **IDirectSound3DBuffer8::SetConeOutsideVolume**, **IDirectSound3DBuffer8::GetConeOrientation**,**IDirectSound3DBuffer8::SetConeOrientation**

IDirectSound3DBuffer8::GetConeOrientation

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

```
HRESULT GetConeOrientation(
  D3DVECTOR * pvOrientation
);
```

Parameters

pvOrientation
Address of a **D3DVECTOR** structure that receives the orientation of the sound projection cone. The vector information represents the center of the sound cone.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

The values returned are not necessarily the same as those set by using the **IDirectSound3DBuffer8::SetConeOrientation** method. DirectSound adjusts orientation vectors so that they are have a magnitude of less than or equal to 1.0.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::SetConeOrientation,
IDirectSound3DBuffer8::SetConeAngles,
IDirectSound3DBuffer8::SetConeOutsideVolume

IDirectSound3DBuffer8::GetConeOutsideVolume

Retrieves the cone outside volume for this sound buffer.

```
HRESULT GetConeOutsideVolume(
  LPLONG plConeOutsideVolume
);
```

Parameters

plConeOutsideVolume
Address of a variable that receives the cone outside volume for this buffer.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

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

Requirements

Header: Declared in dsound.h.

See Also

See Also

IDirectSound3DBuffer8::SetConeOutsideVolume, **IDirectSound3DBuffer8::GetConeOrientation**,**IDirectSound3DBuffer8::SetConeOrientation**

IDirectSound3DBuffer8::GetMaxDistance

Retrieves the maximum distance for this sound buffer.

```
HRESULT GetMaxDistance(
  D3DVALUE * pflMaxDistance
);
```

Parameters

pflMaxDistance
Address of a variable that receives the maximum distance setting.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::GetMinDistance,
IDirectSound3DBuffer8::SetMaxDistance

IDirectSound3DBuffer8::GetMinDistance

Retrieves the minimum distance for this sound buffer.

```
HRESULT GetMinDistance(
  D3DVALUE * pflMinDistance
);
```

Parameters

pflMinDistance
Address of a variable that receives the minimum distance setting.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::SetMinDistance,
IDirectSound3DBuffer8::GetMaxDistance

IDirectSound3DBuffer8::GetMode

Retrieves the operation mode for 3-D sound processing.

```
HRESULT GetMode(
  LPDWORD pdwMode
);
```

Parameters

pdwMode
Address of a variable that receives the mode setting. This value will be one of the following:

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

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

DS3DMODE_NORMAL
Normal processing. This is the default mode.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSound3DBuffer8::GetPosition

Retrieves the sound buffer's position, in distance units. By default, distance units are meters, but the units can be changed by using the **IDirectSound3DListener8::SetDistanceFactor** method.

```
HRESULT GetPosition(
  D3DVECTOR * pvPosition
);
```

Parameters

pvPosition
Address of a **D3DVECTOR** structure that receives the position of the sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSound3DBuffer8::GetVelocity

Retrieves the velocity for this sound buffer. Velocity is measured in units per second. The default unit is one meter, but this can be changed by using the **IDirectSound3DListener8::SetDistanceFactor** method.

```
HRESULT GetVelocity(
  D3DVECTOR * pvVelocity
);
```

Parameters

pvVelocity
Address of a **D3DVECTOR** structure that receives the sound buffer's velocity.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

The default unit of measurement is meters per second, but this can be changed by using the **IDirectSound3DListener8::SetDistanceFactor** method.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::SetPosition, **IDirectSound3DBuffer8::SetVelocity**

IDirectSound3DBuffer8::SetAllParameters

Sets all 3-D sound buffer parameters.

```
HRESULT SetAllParameters(
  LPCDS3DBUFFER pcDs3dBuffer,
  DWORD dwApply
);
```

Parameters

pcDs3dBuffer
Address of a **DS3DBUFFER** structure that describes the 3-D characteristics of the sound buffer.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSound3DBuffer8::SetConeAngles

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

```
HRESULT SetConeAngles(
  DWORD dwInsideConeAngle,
  DWORD dwOutsideConeAngle,
```

DWORD *dwApply*
);

Parameters

dwInsideConeAngle
Value that specifies the inside cone angle, in degrees. This is the angle within which the sound is at its normal volume.

dwOutsideConeAngle
Value that specifies the outside cone angle, in degrees. This is the angle outside of which the sound is at its outside volume.

dwApply
Value that indicates when the setting should be applied. Must be one of the following:

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::SetConeOutsideVolume,
IDirectSound3DBuffer8::SetConeOrientation

IDirectSound3DBuffer8::SetConeOrientation

Sets the orientation of the sound projection cone for this sound buffer. This method has no effect unless the cone angle and cone volume factor have also been set.

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

Parameters

x, y, and *z*
Values that specify the coordinates of the sound cone orientation vector.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::SetConeAngles,
IDirectSound3DBuffer8::SetConeOutsideVolume

IDirectSound3DBuffer8::SetConeOutsideVolume

Sets the cone outside volume for this sound buffer.

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

Parameters

lConeOutsideVolume

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

dwApply

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

DS3D_DEFERRED

Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::SetConeAngles,
IDirectSound3DBuffer8::SetConeOrientation, **IDirectSoundBuffer8::SetVolume**

IDirectSound3DBuffer8::SetMaxDistance

Sets the maximum distance value.

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

Parameters

flMaxDistance
New maximum distance value.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::GetMaxDistance,
IDirectSound3DBuffer8::SetMinDistance

IDirectSound3DBuffer8::SetMinDistance

Sets the minimum distance value.

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

Parameters

flMinDistance
New minimum distance value.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::SetMaxDistance

IDirectSound3DBuffer8::SetMode

Sets the operation mode for 3-D sound processing.

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

Parameters

dwMode

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

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

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

DS3DMODE_NORMAL
: Normal processing. This is the default mode.

dwApply

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

DS3D_DEFERRED
: Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSound3DBuffer8::SetPosition

Sets the sound buffer's position, in distance units. By default, distance units are meters, but the units can be changed by using the **IDirectSound3DListener8::SetDistanceFactor** method.

```
HRESULT SetPosition(
```

D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*,
DWORD *dwApply*
);

Parameters

x, y, and *z*
Values that specify the coordinates of the position vector. DirectSound may adjust these values to prevent floating-point overflow.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSound3DBuffer8::SetVelocity

Sets the velocity of the sound source.

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

Parameters

x, y, and *z*
Values that specify the coordinates of the velocity vector. DirectSound may adjust these values to prevent floating-point overflow.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

Velocity is used only for calculating Doppler effect. It does not change the position of the buffer. For more information, see Doppler Factor.

The default unit of measurement is meters per second, but this can be changed by using the **IDirectSound3DListener8::SetDistanceFactor** method.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DBuffer8::SetPosition, **IDirectSound3DBuffer8::GetVelocity**

IDirectSound3DListener8

Used to retrieve and set parameters that describe a listener's position, orientation, and listening environment in 3-D space.

The interface can be obtained by calling the **IDirectSoundBuffer8::QueryInterface** method on a primary buffer created with the DSBCAPS_CTRL3D flag. Applications that use audiopaths can obtain the interface from the audiopath as in the following code example, where *pAudiopath* is an **IDirectMusicAudioPath8** interface pointer.

```
IDirectSound3DListener8 *pListener;

pAudioPath->GetObjectInPath(0, DMUS_PATH_PRIMARY_BUFFER,
    0, GUID_All_Objects, 0, IID_IDirectSound3DListener,
    (void **)&pListener)));
```

The methods of the **IDirectSound3DListener8** interface can be organized into the following groups:

Batch parameters	**GetAllParameters**
	SetAllParameters
Deferred settings	**CommitDeferredSettings**
Distance factor	**GetDistanceFactor**
	SetDistanceFactor
Doppler factor	**GetDopplerFactor**
	SetDopplerFactor
Orientation	**GetOrientation**
	SetOrientation
Position	**GetPosition**
	SetPosition
Rolloff factor	**GetRolloffFactor**
	SetRolloffFactor
Velocity	**GetVelocity**
	SetVelocity

The **IDirectSound3DListener8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

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

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

Requirements

Header: Declared in dsound.h.

IDirectSound3DListener8::CommitDeferredSettings

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

```
HRESULT CommitDeferredSettings();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

Requirements

Header: Declared in dsound.h.

IDirectSound3DListener8::GetAllParameters

Retrieves the state of the 3-D world and listener.

```
HRESULT GetAllParameters(
  LPDS3DLISTENER pListener
);
```

Parameters

pListener
Address of a **DS3DLISTENER** structure that receives the state of the 3-D world and listener.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::SetAllParameters

IDirectSound3DListener8::GetDistanceFactor

Retrieves the distance factor, which is the number of meters in a vector unit.

```
HRESULT GetDistanceFactor(
  D3DVALUE * pflDistanceFactor
);
```

Parameters

pflDistanceFactor
Address of a variable that receives the distance factor value.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::SetDistanceFactor, Distance Factor

IDirectSound3DListener8::GetDopplerFactor

Retrieves the Doppler effect factor.

```
HRESULT GetDopplerFactor(
  D3DVALUE * pflDopplerFactor
);
```

Parameters

pflDopplerFactor
Address of a variable that receives the Doppler factor value.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::SetDopplerFactor

IDirectSound3DListener8::GetOrientation

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

```
HRESULT GetOrientation(
  D3DVECTOR * pvOrientFront,
  D3DVECTOR * pvOrientTop
);
```

Parameters

pvOrientFront
Address of a **D3DVECTOR** structure that receives the listener's front orientation vector.

pvOrientTop
Address of a **D3DVECTOR** structure that receives the listener's top orientation vector.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

The values returned are not necessarily the same as those set by using the **IDirectSound3DListener8::SetOrientation** method. DirectSound adjusts orientation

vectors so that they are at right angles and have a magnitude of less than or equal to 1.0.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::SetOrientation

IDirectSound3DListener8::GetPosition

Retrieves the listener's position in distance units. By default, these units are meters, but this can be changed by calling the **IDirectSound3DListener8::SetDistanceFactor** method.

```
HRESULT GetPosition(
  D3DVECTOR * pvPosition
);
```

Parameters

pvPosition
Address of a **D3DVECTOR** structure that receives the listener's position vector.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::SetPosition

IDirectSound3DListener8::GetRolloffFactor

Retrieves the rolloff factor, which determines the rate of attenuation over distance.

```
HRESULT GetRolloffFactor(
  D3DVALUE * pflRolloffFactor
);
```

Parameters

pflRolloffFactor
Address of a variable that receives the rolloff factor.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

The rolloff factor has a range of DS3D_MINROLLOFFFACTOR (no rolloff) to DS3D_MAXROLLOFFFACTOR (as currently defined, 10 times the rolloff found in the real world). The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0). For more information, see Rolloff Factor.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::SetRolloffFactor

IDirectSound3DListener8::GetVelocity

Retrieves the listener's velocity.

```
HRESULT GetVelocity(
  D3DVECTOR * pvVelocity
);
```

Parameters

pvVelocity
Address of a **D3DVECTOR** structure that receives the listener's velocity.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

Velocity is used only for calculating Doppler effect. It does not change the listener's position. To move the listener, use the **IDirectSound3DListener8::SetPosition** method.

The default velocity is (0,0,0).

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::SetVelocity

IDirectSound3DListener8::SetAllParameters

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

```
HRESULT SetAllParameters(
  LPCDS3DLISTENER pcListener,
  DWORD dwApply
);
```

Parameters

pcListener

Address of a **DS3DLISTENER** structure that contains information describing all 3-D listener parameters.

dwApply

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

DS3D_DEFERRED

Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

DS3D_IMMEDIATE

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::GetAllParameters

IDirectSound3DListener8::SetDistanceFactor

Sets the distance factor, which is the number of meters in a vector unit.

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

Parameters

flDistanceFactor
Value that specifies the distance factor.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

The distance factor has a range of DS3D_MINDISTANCEFACTOR to DS3D_MAXDISTANCEFACTOR, defined in Dsound.h as FLT_MIN and FLT_MAX respectively. The default value is DS3D_DEFAULTDISTANCEFACTOR, or 1.0.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::GetDistanceFactor, Distance Factor

IDirectSound3DListener8::SetDopplerFactor

Sets the Doppler effect factor.

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

Parameters

flDopplerFactor
Value that specifies the Doppler factor.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::GetDopplerFactor

IDirectSound3DListener8::SetOrientation

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

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

Parameters

xFront, yFront, zFront
Values that specify the coordinates of the front orientation vector.

xTop, yTop, zTop
Values that specify the coordinates of the top orientation vector.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

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

The top vector must be at right angles to the front vector. If necessary, DirectSound adjusts the front vector after setting the top vector.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::GetOrientation

IDirectSound3DListener8::SetPosition

Sets the listener's position, in distance units. By default, these units are meters, but this can be changed by calling the **IDirectSound3DListener8::SetDistanceFactor** method.

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

Parameters

x, y, and *z*
Values that specify the coordinates of the listener's position vector. Note that DirectSound may adjust these values to prevent floating-point overflow.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::GetPosition

IDirectSound3DListener8::SetRolloffFactor

Sets the rolloff factor, which determines the rate of attenuation over distance.

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

Parameters

flRolloffFactor
Value that specifies the rolloff factor.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

The rolloff factor has a range of DS3D_MINROLLOFFFACTOR (no rolloff) to DS3D_MAXROLLOFFFACTOR (as currently defined, 10 times the rolloff found in the real world). The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0). For more information, see Rolloff Factor.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::GetRolloffFactor

IDirectSound3DListener8::SetVelocity

Sets the listener's velocity.

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

Parameters

x, y, and *z*
Values that specify the coordinates of the listener's velocity vector. DirectSound may adjust these values to prevent floating-point overflow.

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

DS3D_DEFERRED
Settings are not applied until the application calls the **IDirectSound3DListener8::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

Velocity is used only for Doppler effects. It does not actually move the listener. To change the listener's position, use the **IDirectSound3DListener8::SetPosition** method. The default velocity is (0,0,0).

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound3DListener8::GetVelocity

IDirectSoundBuffer8

Used to manage sound buffers.

The **IDirectSoundBuffer8** interface supersedes **IDirectSoundBuffer** and adds new methods.

To obtain the interface, use the **IDirectSound8::CreateSoundBuffer** method to retrieve **IDirectSoundBuffer**, and then pass IID_IDirectSoundBuffer8 to **IDirectSoundBuffer::QueryInterface**.

For primary sound buffers, you must use the **IDirectSoundBuffer** interface. **IDirectSoundBuffer8** is not available. **IDirectSoundBuffer** is not documented separately. For documentation, see the corresponding **IDirectSoundBuffer8** methods.

Not all methods of **IDirectSoundBuffer** are valid for primary buffers. For example, **SetCurrentPosition** will fail. See the reference topics for individual methods.

The **IDirectSoundBuffer8** methods can be organized into the following groups:

Effects	**GetObjectInPath**
	SetFX
Information	**GetCaps**
	GetFormat
	GetStatus
	SetFormat
Memory management	**AcquireResources**
	Initialize
	Restore
Play management	**GetCurrentPosition**
	Lock
	Play

	SetCurrentPosition
	Stop
	Unlock
Sound management	**GetFrequency**
	GetPan
	GetVolume
	SetFrequency
	SetPan
	SetVolume

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The following table shows which methods are supported for buffer objects obtained from an audiopath; that is, buffers created by a DirectMusic performance. Mix-in buffers accept sends from other buffers.All other buffers in the audiopath are sink-in buffers, which means that they accept data only from the synthesizer sink.

IDirectSoundBuffer8 method	**Mix-in**	**Sink-in**
AcquireResources		
GetCaps	Yes	Yes
GetCurrentPosition		
GetFormat	Yes	Yes
GetFrequency		
GetObjectInPath	Yes	Yes
GetPan	Yes	Yes
GetStatus	Yes	Yes
GetVolume	Yes	Yes
Initialize		
Lock		
Play	Yes	
Restore		
SetCurrentPosition		
SetFormat		
SetFrequency		
SetFX	Yes	Yes
SetPan	Yes	Yes

SetVolume	Yes	Yes
Stop	Yes	
Unlock		

The **LPDIRECTSOUNDBUFFER8** type is defined as a pointer to the **IDirectSoundBuffer** interface:

```
typedef struct IDirectSoundBuffer8    *LPDIRECTSOUNDBUFFER8;
```

Requirements

Header: Declared in dsound.h.

IDirectSoundBuffer8::AcquireResources

Allocates resources for a buffer that was created with the DSBCAPS_LOCDEFER flag in the **DSBUFFERDESC** structure.

```
HRESULT AcquireResources(
  DWORD dwFlags,
  DWORD dwEffectsCount,
  LPDWORD pdwResultCodes
);
```

Parameters

dwFlags

Flags specifying how resources are to be allocated for a buffer created with the DSBCAPS_LOCDEFER flag. For a list of values, see the voice management and voice allocation flags for **IDirectSoundBuffer8::Play**.

dwEffectsCount

Value that specifies the number of elements in the *pdwResultFlags* array, or 0 if *pdwResultCodes* is NULL.

pdwResultCodes

Address of an array of **DWORD** variables that receives information about the effects associated with the buffer. This array must contain one element for each effect that was assigned to the buffer by **IDirectSoundBuffer8::SetFX**. For each effect, one of the following values is returned.

DSFXR_LOCHARDWARE
Effect is instantiated in hardware.

DSFXR_LOCSOFTWARE
Effect is instantiated in software.

DSFXR_FAILED
No effect was created because resources weren't available.

DSFXR_PRESENT

The effect is available but was not created because the method failed for some other reason.

DSFXR_UNKNOWN
No effect was created because the effect isn't registered on the system.

This parameter can be NULL if result codes are not wanted.

Return Values

If the method succeeds, the return value is DS_OK or DS_INCOMPLETE.

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

DSERR_CONTROLUNAVAIL

DSERR_INVALIDCALL

DSERR_INVALIDPARAM

Remarks

Normally, buffers created with DSBCAPS_LOCDEFER are not allocated resources until **IDirectSoundBuffer8::Play** is called.
IDirectSoundBuffer8::AcquireResources can be used to allocate resources for deferred buffers. By doing so, the application can retrieve information about effects processing and set effect parameters before the buffer is played.

If the method fails, check the values in *pdwResultCodes* to determine which effects caused the failure.

A buffer with acquired resources that is not yet playing is not a candidate for premature termination by the voice management flags passed to the **Play** method.

Resources that have been acquired by **AcquireResources** are released when playback is stopped.

If the method is called on a buffer on which it has already been called, the status of the effects is returned but no additional resources are allocated.

The *dwEffectsCount* parameter to this function must be the same as the one passed in the call to **IDirectSoundBuffer8::SetFX**.

If an attempt is made to acquire resources for a buffer with the DSBCAPS_LOCHARDWARE flag on a system where hardware acceleration is not available, the method fails with either DSERR_CONTROLUNAVAIL or DSERR_INVALIDCALL, depending on the operating system.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX, **DSBCAPS**

IDirectSoundBuffer8::GetCaps

Retrieves the capabilities of the DirectSoundBuffer object.

```
HRESULT GetCaps(
  LPDSBCAPS pDSBufferCaps
);
```

Parameters

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

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

The **DSBCAPS** structure contains similar information to the **DSBUFFERDESC** structure passed to the **IDirectSound8::CreateSoundBuffer** method, with some additional information. This additional information can include the buffer's location, either in hardware or software, and performance measures.

The flags specified in the **dwFlags** member of the **DSBCAPS** structure are the same flags used by the **DSBUFFERDESC** structure. The only difference is that in the **DSBCAPS** structure, either DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE will be specified according to the location of the buffer memory. In the **DSBUFFERDESC** structure, these flags are optional and, depending on which flag is specified, force the buffer to be located in either hardware or software.

Requirements

Header: Declared in dsound.h.

See Also

DSBCAPS, **DSBUFFERDESC**, **IDirectSound8::CreateSoundBuffer**

IDirectSoundBuffer8::GetCurrentPosition

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

```
HRESULT GetCurrentPosition(
  LPDWORD pdwCurrentPlayCursor,
  LPDWORD pdwCurrentWriteCursor
);
```

Parameters

pdwCurrentPlayCursor
Address of a variable that receives the offset, in bytes, of the play cursor. This parameter can be NULL if the value is not wanted.

pdwCurrentWriteCursor
Address of a variable that receives the offset, in bytes, of the write cursor. This parameter can be NULL if the value is not wanted.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The write cursor is the point in the buffer ahead of which it is safe to write data to the buffer. Data should not be written to the part of the buffer after the play cursor and before the write cursor. For more information, see Play and Write Cursors.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetCurrentPosition

IDirectSoundBuffer8::GetFormat

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

```
HRESULT GetFormat(
  LPWAVEFORMATEX pwfxFormat,
  DWORD dwSizeAllocated,
  LPDWORD pdwSizeWritten
);
```

Parameters

pwfxFormat
Address of a **WAVEFORMATEX** structure that receives a description of the sound data in the buffer. To retrieve the buffer size needed to contain the format description, specify NULL. In this case the variable at *pdwSizeWritten* receives the size of the structure needed to receive the data.

dwSizeAllocated
Size, in bytes, of the structure at *pwfxFormat*. If *pwfxFormat* is not NULL, this value must be equal to or greater than the size of the expected data.

pdwSizeWritten
Address of a variable that receives the number of bytes written to the structure at *pwfxFormat*. This parameter can be NULL.

Return Values

If the method succeeds, the return value is DS_OK.

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

Remarks

The **WAVEFORMATEX** structure can have a variable length that depends on the format. Before retrieving the format description, the application should query the DirectSoundBuffer object for the size of the format by calling this method and specifying NULL for the *pwfxFormat* parameter. The necessary size of the structure is returned in the *pdwSizeWritten* parameter. The application can then allocate sufficient memory and call **IDirectSoundBuffer8::GetFormat** again to retrieve the format description.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFormat

IDirectSoundBuffer8::GetFrequency

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

```
HRESULT GetFrequency(
  LPDWORD pdwFrequency
);
```

Parameters

pdwFrequency
Address of a variable that receives the frequency at which the audio buffer is being played, in hertz.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The frequency value for software buffers will be in the range of DSBFREQUENCY_MIN to DSBFREQUENCY_MAX. These values are defined in Dsound.h as 100 and 100,000 respectively. Hardware buffers can return other values.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFrequency

IDirectSoundBuffer8::GetObjectInPath

Retrieves an interface for an effect object associated with the buffer.

```
HRESULT GetObjectInPath(
  REFGUID rguidObject,
  DWORD dwIndex,
  REFGUID rguidInterface,
  LPVOID * ppObject
);
```

Parameters

rguidObject
Value of type **REFGUID** that specifies the unique class identifier of the object being searched for, such as GUID_DSFX_STANDARD_ECHO. Set this parameter to GUID_All_Objects to search for objects of any class.

dwIndex

Value that specifies the index of the object within objects of that class in the path. See Remarks.

rguidInterface
Value that specifies the unique identifier of the desired interface. See Remarks.

ppObject
Address of a variable that receives the desired interface pointer.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_OBJECTNOTFOUND
E_NOINTERFACE

Remarks

Any DMO that has been set on a buffer by using **IDirectSoundBuffer8::SetFX** can be retrieved, even it has not been allocated resources.

The following interfaces can be retrieved for the various DMOs supplied with DirectX.

rguidInterface	****ppObject***
IID_IDirectSoundFXGargle8	**IDirectSoundFXGargle8**
IID_IDirectSoundFXChorus8	**IDirectSoundFXChorus8**
IID_IDirectSoundFXFlanger8	**IDirectSoundFXFlanger8**
IID_IDirectSoundFXEcho8	**IDirectSoundFXEcho8**
IID_IDirectSoundFXDistortion8	**IDirectSoundFXDistortion8**
IID_IDirectSoundFXCompressor8	**IDirectSoundFXCompressor8**
IID_IDirectSoundFXParamEq8	**IDirectSoundFXParamEq8**
IID_IDirectSoundFXWavesReverb8	**IDirectSoundFXWavesReverb8**
IID_IDirectSoundFXI3DL2Reverb8	**IDirectSoundFXI3DL2Reverb8**

In addition, the following interfaces are available for any of the standard DMOs. For information on these interfaces, see the DirectShow Help.

rguidInterface	****ppObject***
IID_IMediaObject	**IMediaObject**
IID_IMediaObjectInPlace	**IMediaObjectInPlace**
IID_IMediaParams	**IMediaParams**

The value in *dwIndex* is the index of the object within the array of effects passed to **IDirectSoundBuffer8::SetFX**. This is not necessarily the actual position of the object in the effects chain, because some effects might not have been created.

An object is returned solely on the basis of whether it matches *rguidObject* and *dwIndex*. It is up to the application to ensure that *rguidInterface* specifies an interface that can be expected to be found on the object.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXChorus8, **IDirectSoundFXCompressor8**, **IDirectSoundFXDistortion8**, **IDirectSoundFXEcho8**, **IDirectSoundFXFlanger8**, **IDirectSoundFXParamEq8**, **IDirectSoundFXI3DL2Reverb8**

IDirectSoundBuffer8::GetPan

Retrieves the relative volume between the left and right audio channels.

```
HRESULT GetPan(
  LPLONG plPan
);
```

Parameters

plPan

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

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_CONTROLUNAVAIL

DSERR_INVALIDPARAM

DSERR_PRIOLEVELNEEDED

Remarks

The returned value is measured in hundredths of a decibel (dB), in the range of DSBPAN_LEFT to DSBPAN_RIGHT. These values are currently defined in Dsound.h as –10,000 and 10,000 respectively. The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER, defined as

zero. This value of 0 in the *plPan* parameter means that both channels are at full volume (they are attenuated by 0 decibels). At any setting other than DSBPAN_CENTER, one of the channels is at full volume and the other is attenuated.

A pan of –2173 means that the left channel is at full volume and the right channel is attenuated by 21.73 dB. Similarly, a pan of 870 means that the left channel is attenuated by 8.7 dB and the right channel is at full volume. A pan of DSBPAN_LEFT means that the right channel is silent and the sound is all the way to the left, while a pan of DSBPAN_RIGHT means that the left channel is silent and the sound is all the way to the right.

The pan control acts cumulatively with the volume control.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::GetVolume, **IDirectSoundBuffer8::SetPan**, **IDirectSoundBuffer8::SetVolume**

IDirectSoundBuffer8::GetStatus

Retrieves the status of the sound buffer.

```
HRESULT GetStatus(
  LPDWORD pdwStatus
);
```

Parameters

pdwStatus

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

DSBSTATUS_BUFFERLOST
: The buffer is lost and must be restored before it can be played or locked.

DSBSTATUS_LOOPING
: The buffer is being looped. If this value is not set, the buffer will stop when it reaches the end of the sound data. This value is returned only in combination with DSBSTATUS_PLAYING.

DSBSTATUS_PLAYING
: The buffer is playing. If this value is not set, the buffer is stopped.

DSBSTATUS_LOCSOFTWARE
: The buffer is playing in software. Set only for buffers created with the DSBCAPS_LOCDEFER flag.

DSBSTATUS_LOCHARDWARE

The buffer is playing in hardware. Set only for buffers created with the DSBCAPS_LOCDEFER flag.

DSBSTATUS_TERMINATED
The buffer was prematurely terminated by the voice manager and is not playing. Set only for buffers created with the DSBCAPS_LOCDEFER flag.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundBuffer8::GetVolume

Retrieves the volume of the buffer.

```
HRESULT GetVolume(
  LPLONG plVolume
);
```

Parameters

plVolume
Address of a variable that receives the attenuation, in hundredths of a decibel. See Remarks.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The return value is between DSBVOLUME_MAX and DSBVOLUME_MIN. These values are defined as 0 and –10,000, respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the sound. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for practical purposes, is silence.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetVolume

IDirectSoundBuffer8::Initialize

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

```
HRESULT Initialize(
  LPDIRECTSOUND pDirectSound,
  LPCDSBUFFERDESC pcDSBufferDesc
);
```

Parameters

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

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

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_ALREADYINITIALIZED

Remarks

Because the **IDirectSound8::CreateSoundBuffer** method calls **IDirectSoundBuffer8::Initialize** internally, it is not needed for the current release of DirectSound. This method is provided for future extensibility.

Requirements

Header: Declared in dsound.h.

See Also

DSBUFFERDESC, **IDirectSound8::CreateSoundBuffer**, **IDirectSoundBuffer8**

IDirectSoundBuffer8::Lock

Readies all or part of the buffer for a data write and returns pointers to which data can be written.

```
HRESULT Lock(
  DWORD dwOffset,
  DWORD dwBytes,
  LPVOID * ppvAudioPtr1,
  LPDWORD pdwAudioBytes1,
  LPVOID * ppvAudioPtr2,
  LPDWORD pdwAudioBytes2,
  DWORD dwFlags
);
```

Parameters

dwOffset
Value that specifies the offset, in bytes, from the start of the buffer to the point where the lock begins. This parameter is ignored if DSBLOCK_FROMWRITECURSOR is specified in the *dwFlags* parameter.

dwBytes
Value that specifies the size, in bytes, of the portion of the buffer to lock. The buffer is conceptually circular, so this number can exceed the number of bytes between *dwOffset* and the end of the buffer.

ppvAudioPtr1
Address of a variable that receives a pointer to the first locked part of the buffer.

pdwAudioBytes1
Address of a variable that receives the number of bytes in the block at *ppvAudioPtr1*. If this value is less than *dwBytes*, the lock has wrapped and *ppvAudioPtr2* points to a second block of data at the beginning of the buffer .

ppvAudioPtr2
Address of a variable that receives a pointer to the second locked part of the capture buffer. If NULL is returned, the *ppvAudioPtr1* parameter points to the entire locked portion of the capture buffer.

pdwAudioBytes2
Address of a variable that receives the number of bytes in the block at *ppvAudioPtr2*. If *ppvAudioPtr2* is NULL, this value is zero.

dwFlags
Flags modifying the lock event. The following flags are defined:

DSBLOCK_FROMWRITECURSOR
Start the lock at the write cursor. The *dwOffset* parameter is ignored.

DSBLOCK_ENTIREBUFFER
Lock the entire buffer. The *dwBytes* parameter is ignored.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

This method accepts an offset and a byte count, and returns two write pointers and their associated sizes. If the locked portion does not extend to the end of the buffer and wrap to the beginning, the second pointer, *ppvAudioBytes2*, receives NULL. If the lock does wrap, *ppvAudioBytes2* points to the beginning of the buffer.

If the application passes NULL for the *ppvAudioPtr2* and *pdwAudioBytes2* parameters, the lock extends no further than the end of the buffer and does not wrap.

After writing data to the pointers returned by this method, the application must immediately call **IDirectSoundBuffer8::Unlock** to notify DirectSound that the data is ready for playback. Failure to do so can cause audio breakup or silence on some sound device configurations.

This method returns write pointers only. The application should not try to read sound data from this pointer, because the data might not be valid. For example, if the buffer is located in on-card memory, the pointer might be an address to a temporary buffer in system memory. When **IDirectSoundBuffer8::Unlock** is called, the contents of this temporary buffer are transferred to the on-card memory.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::GetCurrentPosition

IDirectSoundBuffer8::Play

Causes the sound buffer to play, starting at the play cursor.

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

Parameters

dwReserved1
Reserved. Must be 0.

dwPriority
Priority for the sound, used by the voice manager when assigning hardware mixing resources. The lowest priority is 0, and the highest priority is 0xFFFFFFFF. If the buffer was not created with the DSBCAPS_LOCDEFER flag, this value must be 0.

dwFlags
Flags specifying how to play the buffer. The following flags are defined:

DSBPLAY_LOOPING
After the end of the audio buffer is reached, play restarts at the beginning of the buffer. Play continues until explicitly stopped. This flag must be set when playing a primary sound buffer.

Voice allocation flags
The voice allocation flags are valid only for buffers created with the DSBCAPS_LOCDEFER flag. One of the following flags can be used to force the processing of the sound into hardware or software. If neither DBSPLAY_LOCHARDWARE nor DBSPLAY_LOCSOFTWARE is set, the sound is played in either software or hardware, depending on the availability of resources at the time the method is called. See Remarks.

DSBPLAY_LOCHARDWARE
Play this voice in a hardware buffer only. If the hardware has no available voices and no voice management flags are set, the call to **IDirectSoundBuffer8::Play** fails. This flag cannot be combined with DSBPLAY_LOCSOFTWARE.

DSBPLAY_LOCSOFTWARE
Play this voice in a software buffer only. This flag cannot be combined with DSBPLAY_LOCHARDWARE or any voice management flag.

Voice management flags
The voice management flags are valid only for buffers created with the DSBCAPS_LOCDEFER flag, and are used for sounds that are to play in hardware. These flags enable hardware resources that are already in use to be yielded to the current sound. Only buffers created with the DSBCAPS_LOCDEFER flag are candidates for premature termination. See Remarks.

DSBPLAY_TERMINATEBY_TIME
If the hardware has no available voices, a currently playing nonlooping buffer will be stopped to make room for the new buffer. The buffer prematurely terminated is the one with the least time left to play.

DSBPLAY_TERMINATEBY_DISTANCE
If the hardware has no available voices, a currently playing buffer will be stopped to make room for the new buffer. The buffer prematurely terminated will be selected from buffers that have the buffer's DSBCAPS_

MUTE3DATMAXDISTANCE flag set and are beyond their maximum distance. If there are no such buffers, the method fails.

DSBPLAY_TERMINATEBY_PRIORITY
If the hardware has no available voices, a currently playing buffer will be stopped to make room for the new buffer. The buffer prematurely terminated will be the one with the lowest priority as set by the *dwPriority* parameter passed to **IDirectSoundBuffer8::Play** for the buffer.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

If **IDirectSound8::SetCooperativeLevel** has not been called, the method returns DS_OK, but no sound will be produced until a cooperative level has been set.

If the buffer specified in the method is already playing, the call to the method succeeds and the buffer continues to play. However, the flags defined in the most recent call supersede flags defined in previous calls.

When called on the primary buffer, this method causes the buffer to start playing to the sound device. If the application has set the DSSCL_WRITEPRIMARY cooperative level, any audio data put in the primary buffer by the application is sent to the sound device. Under any other cooperative level, the primary buffer plays silence if no secondary buffers are playing. Primary buffers must be played with the DSBPLAY_LOOPING flag set. For more information, see Playing the Primary Buffer Continuously.

If the method is called with a voice allocation or voice management flag set on a buffer that was not created with the DSBCAPS_LOCDEFER flag, the call fails with DSERR_INVALIDPARAM.

DSBPLAY_TERMINATEBY_TIME and DSBPLAY_TERMINATEBY_DISTANCE cannot be combined, but either may be combined with DSBPLAY_TERMINATEBY_PRIORITY, in which case the DSBPLAY_TERMINATEBY_TIME or DSBPLAY_TERMINATEBY_DISTANCE flag is used to determine which buffer should be terminated in the event of a priority tie.

The following table shows the behavior of the method under various combinations of the voice allocation and voice management flags when no free hardware voices are available.

	DSBPLAY_LOCHARDWARE	Neither DSBPLAY_LOCHARDWARE nor DSBPLAY_LOCSOFTWARE	DSBPLAY_LOCSOFTWARE
DSBPLAY_TERMINATEBY_TIME	Sound with least time left to play is terminated and the new sound plays on the released voice.	Sound with least time left to play is terminated and the new sound plays on the released voice.	New sound plays in software.
DSBPLAY_TERMINATEBY_DISTANCE	If any sounds currently playing in hardware are beyond their maximum distance and have the DSBCAPS_MUTE3DATMAXDISTANCE flag set, one of them is terminated and the new sound plays in hardware. Otherwise, the call fails.	If any sounds currently playing in hardware are beyond their maximum distance and have the DSBCAPS_MUTE3DATMAXDISTANCE flag set, one of them is terminated and the new sound plays in hardware. Otherwise, the new sound plays in software.	New sound plays in software.
DSBPLAY_TERMINATEBY_PRIORITY	If the new sound's priority is higher than or equal to that of any sound currently playing in hardware, one of the lowest-priority sounds is terminated and the new sound plays in hardware. Otherwise, the call fails.	If the new sound's priority is higher than or equal to that of any sound currently playing in hardware, one of the lowest-priority sounds is terminated and the new sound plays in hardware. Otherwise, the new sound plays in software.	New sound plays in software.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8, **IDirectSound8::SetCooperativeLevel**

IDirectSoundBuffer8::Restore

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

HRESULT Restore();

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

If the application does not have the input focus, **IDirectSoundBuffer8::Restore** might not succeed. For example, if the application with the input focus has the DSSCL_WRITEPRIMARY cooperative level, no other application will be able to restore its buffers. Similarly, an application with the DSSCL_WRITEPRIMARY cooperative level must have the input focus to restore its primary sound buffer.

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

The application can receive notification that a buffer is lost when it specifies that buffer in a call to the **IDirectSoundBuffer8::Lock** or **IDirectSoundBuffer8::Play** method. These methods return DSERR_BUFFERLOST to indicate a lost buffer. The **IDirectSoundBuffer8::GetStatus** method can also be used to retrieve the status of the sound buffer and test for the DSBSTATUS_BUFFERLOST flag.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::Lock, **IDirectSoundBuffer8::Play**, **IDirectSoundBuffer8::GetStatus**

IDirectSoundBuffer8::SetCurrentPosition

Sets the position of the play cursor, which is the point at which the next byte of data is read from the buffer.

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

Parameters

dwNewPosition
Offset of the play cursor, in bytes, from the beginning of the buffer.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

This method cannot be called on primary sound buffers.

If the buffer is playing, the cursor immediately moves to the new position and play continues from that point. If it is not playing, playback will begin from the new position the next time the **IDirectSoundBuffer8::Play** method is called.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::GetCurrentPosition, **IDirectSoundBuffer8::Play**

IDirectSoundBuffer8::SetFormat

Sets the format of the primary sound buffer for the application. Whenever this application has the input focus, DirectSound will set the primary buffer to the specified format.

Since primary sound buffers do not support the **IDirectSoundBuffer8** interface, this method must be called on **IDirectSoundBuffer**.

```
HRESULT SetFormat(
  LPCWAVEFORMATEX pcfxFormat
);
```

Parameters

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

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_BADFORMAT
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY
DSERR_PRIOLEVELNEEDED
DSERR_UNSUPPORTED

Remarks

The format of the primary buffer should be set before secondary buffers are created.

The method fails if the application has the DSSCL_NORMAL cooperative level.

If the application is using DirectSound at the DSSCL_WRITEPRIMARY cooperative level, the buffer must be stopped before **IDirectSoundBuffer8::SetFormat** is called. If the format is not supported, the method fails.

If the cooperative level is DSSCL_PRIORITY or DSSCL_EXCLUSIVE, DirectSound stops the primary buffer, changes the format, and restarts the buffer. The method succeeds even if the hardware does not support the requested format; DirectSound sets the buffer to the closest supported format. To determine whether this has happened, an application can call the **IDirectSoundBuffer8::GetFormat** method for the primary buffer and compare the result with the format that was requested with the **SetFormat** method.

This method is not available for secondary sound buffers. If a new format is required, the application must create a new DirectSoundBuffer object.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8, **IDirectSoundBuffer8::GetFormat**

IDirectSoundBuffer8::SetFrequency

Sets the frequency at which the audio samples are played.

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

Parameters

dwFrequency

New frequency, in hertz (Hz), at which to play the audio samples. The value must be in the range DSBFREQUENCY_MIN to DSBFREQUENCY_MAX. These values are currently defined in Dsound.h as 100 and 100,000 respectively.

If the value is DSBFREQUENCY_ORIGINAL, the frequency is reset to the default value in the buffer format. This format is specified in the **IDirectSound8::CreateSoundBuffer** method.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_CONTROLUNAVAIL

DSERR_GENERIC

DSERR_INVALIDPARAM

DSERR_PRIOLEVELNEEDED

Remarks

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

This method is not valid for primary sound buffers.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound8::CreateSoundBuffer, **IDirectSoundBuffer8::GetFrequency**, **IDirectSoundBuffer8::Play**, **IDirectSoundBuffer8::SetFormat**

IDirectSoundBuffer8::SetFX

Enables effects on a buffer. The buffer must not be playing or locked.

For this method to succeed, **CoInitialize** must have been called. This is done automatically in applications that create DirectSound by using **IDirectMusicPerformance8::InitAudio**. However, applications that create DirectSound by using **DirectSoundCreate8** must also call **CoInitialize**.

```
HRESULT SetFX(
  DWORD dwEffectsCount,
  LPDSEFFECTDESC pDSFXDesc,
  LPDWORD pdwResultCodes
);
```

Parameters

dwEffectsCount
Value that specifies the number of elements in the *pDSFXDesc* and *pdwResultCodes* arrays. If this value is 0, *pDSFXDesc* and *pdwResultCodes* must both be NULL. Set to 0 to remove all effects from the buffer.

pDSFXDesc
Address of an array of **DSEFFECTDESC** structures, of size *dwEffectsCount*, that specifies the effects wanted on the buffer. Must be NULL if *dwEffectsCount* is 0.

pdwResultCodes
Address of an array of **DWORD** elements, of size *dwEffectsCount*. Each element receives one of the following values indicating the result of the attempt to create the corresponding effect in the *pDSFXDesc* array.

DSFXR_LOCHARDWARE
Effect is instantiated in hardware.

DSFXR_LOCSOFTWARE
Effect is instantiated in software.

DSFXR_UNALLOCATED
Effect has not yet been assigned to hardware nor software. This value is returned if the buffer was created with the DSBCAPS_LOCDEFER flag in **DSBUFFERDESC**.

DSFXR_FAILED
No effect was created because resources weren't available.

DSFXR_PRESENT
The effect is available but was not created because one of the other effects requested could not be created. If any of the effects requested cannot be created, none of them are, and the call fails.

DSFXR_UNKNOWN
The effect is not registered on the system, and the method failed as a result.

This parameter can be NULL is no result values are wanted, and must be NULL if *dwEffectsCount* is 0

Return Values

If the method succeeds, the return value is DS_OK or DS_INCOMPLETE.

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

CO_E_NOTINITIALIZED
DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_INVALIDCALL
DSERR_NOINTERFACE
DSERR_PRIOLEVELNEEDED

In addition, the method can return an error value from DMO methods, including **IMediaObject::SetInputType** and **IMediaObject::SetOutputType**. See the DirectShow documentation for possible error values.

Remarks

If the method fails, the value for each effect in *pdwResultCodes* is either DSFXF_PRESENT or DSFXR_UNKNOWN. Check these values to determine which effects caused the failure.

For the method to succeed, the buffer must have been created with the DSBCAPS_CTRLFX flag and must not be playing or locked.

If the method returns DSERR_NOINTERFACE or another COM error, check the result code array for DSFXR_PRESENT or DSFXR_UNKNOWN to ascertain which effect caused the error. If the method returns DSERR_INVALIDPARAM, check the result codes for DSFXR_FAILED to see which effects failed to acquire resources.

If the buffer is part of an audiopath, the audiopath must be inactive. See **IDirectMusicAudioPath8::Activate**.

An effect must be set on a buffer before the effect interface can be obtained. To obtain the effect interface, use **IDirectSoundBuffer8::GetObjectInPath**.

Requirements

Header: Declared in dsound.h.

See Also

DSBCAPS, **IDirectSoundBuffer8::AcquireResources**

IDirectSoundBuffer8::SetPan

Sets the relative volume of the left and right channels.

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

Parameters

lPan
Relative volume between the left and right channels.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The value in *lPan* is measured in hundredths of a decibel (dB), in the range of DSBPAN_LEFT to DSBPAN_RIGHT. These values are currently defined in Dsound.h as –10,000 and 10,000 respectively. The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER, defined as zero. This value of 0 in the *plPan* parameter means that both channels are at full volume (they are attenuated by 0 decibels). At any setting other than DSBPAN_CENTER, one of the channels is at full volume and the other is attenuated.

A pan of –2173 means that the left channel is at full volume and the right channel is attenuated by 21.73 dB. Similarly, a pan of 870 means that the left channel is attenuated by 8.7 dB and the right channel is at full volume. A pan of DSBPAN_LEFT means that the right channel is silent and the sound is all the way to the left, while a pan of DSBPAN_RIGHT means that the left channel is silent and the sound is all the way to the right.

The pan control acts cumulatively with the volume control.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::GetPan, **IDirectSoundBuffer8::GetVolume**, **IDirectSoundBuffer8::SetVolume**

IDirectSoundBuffer8::SetVolume

Sets the sound volume.

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

Parameters

lVolume
New volume requested for this sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The volume is specified in hundredths of decibels (dB). Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are currently defined in Dsound.h as 0 and –10,000 respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the stream. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for all practical purposes, is silence. Currently DirectSound does not support amplification.

Requirements

Header: Declared in dsound.h.

See Also

l**DirectSoundBuffer8::GetPan**, **IDirectSoundBuffer8::GetVolume**, **IDirectSoundBuffer8::SetPan**

IDirectSoundBuffer8::Stop

Causes the sound buffer to stop playing.

HRESULT Stop();

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

For secondary sound buffers, **IDirectSoundBuffer8::Stop** sets the play cursor to the sample that follows the last sample played. This means that when the **IDirectSoundBuffer8::Play** method is next called on the buffer, it will continue playing where it left off.

For primary sound buffers, if an application has the DSSCL_WRITEPRIMARY level, this method will stop the buffer and reset the play cursor to 0 (the beginning of the buffer). This is necessary because the primary buffers on most sound cards can play only from the beginning of the buffer.

However, if **IDirectSoundBuffer8::Stop** is called on a primary buffer and the application has a cooperative level other than DSSCL_WRITEPRIMARY, this method simply reverses the effects of **IDirectSoundBuffer8::Play**. It configures the primary buffer to stop if no secondary buffers are playing. If other buffers are playing in this or other applications, the primary buffer will not actually stop until they are stopped. This method is useful because playing the primary buffer consumes processing overhead even if the buffer is playing sound data with the amplitude of 0 decibels.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::Play

IDirectSoundBuffer8::Unlock

Releases a locked sound buffer.

```
HRESULT Unlock(
  LPVOID pvAudioPtr1,
  DWORD dwAudioBytes1,
  LPVOID pvAudioPtr2,
  DWORD dwAudioBytes2
);
```

Parameters

pvAudioPtr1
Address of the value retrieved in the *ppvAudioPtr1* parameter of the **IDirectSoundBuffer8::Lock** method.

dwAudioBytes1
Value that specifies the number of bytes written to the portion of the buffer at *pvAudioPtr1*. See Remarks.

pvAudioPtr2
Address of the value retrieved in the *ppvAudioPtr2* parameter of the **IDirectSoundBuffer8::Lock** method.

dwAudioBytes2
Value that specifies the number of bytes written to the portion of the buffer at *pvAudioPtr2*. See Remarks.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

An application must pass both pointers, *pvAudioPtr1* and *pvAudioPtr2*, returned by the **IDirectSoundBuffer8::Lock** method to ensure the correct pairing of **IDirectSoundBuffer8::Lock** and **IDirectSoundBuffer8::Unlock**. The second pointer is needed even if nothing was written to the second pointer.

The values in *dwAudioBytes1* and *dwAudioBytes2* must specify the number of bytes actually written to each part of the buffer, which might be less than the size of the lock. DirectSound uses these values to determine how much data to commit to the device.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::GetCurrentPosition, **IDirectSoundBuffer8::Lock**

IDirectSoundCapture8

The methods of the **IDirectSoundCapture8** interface are used to create sound capture buffers.

The interface is obtained by using the **DirectSoundCaptureCreate8** function or **DirectSoundFullDuplexCreate8** function, or , or by using **CoCreateInstance** or **CoCreateInstanceEx** to create an object of class CLSID_DirectSoundCapture8.

IDirectSoundCapture8 is a define for **IDirectSoundCapture**. The two interface names are interchangeable. However, objects supporting this interface can have different functionality, depending on their class. For more information, see **DirectSoundCaptureCreate8** and **IDirectSoundCapture8::CreateCaptureBuffer**.

This reference section gives information on the following methods of the **IDirectSoundCapture8** interface:

Creation	**CreateCaptureBuffer**
	Initialize
Capabilities	**GetCaps**

Like all COM interfaces, the **IDirectSoundCapture8** interface inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTSOUNDCAPTURE** type is defined as a pointer to the **IDirectSoundCapture** interface:

```
typedef struct IDirectSoundCapture   *LPDIRECTSOUNDCAPTURE;
```

Requirements

Header: Declared in dsound.h.

IDirectSoundCapture8::CreateCaptureBuffer

Creates a capture buffer.

```
HRESULT CreateCaptureBuffer(
  LPCDSCBUFFERDESC pcDSCBufferDesc,
  LPDIRECTSOUNDCAPTUREBUFFER * ppDSCBuffer,
  LPUNKNOWN pUnkOuter
);
```

Parameters

pcDSCBufferDesc
Pointer to a **DSCBUFFERDESC** structure containing values for the capture buffer being created.

ppDSCBuffer
Address of a variable that receives an **IDirectSoundCaptureBuffer** interface pointer. Use **QueryInterface** to obtain **IDirectSoundCaptureBuffer8**. See Remarks.

pUnkOuter
Address of the controlling object's **IUnknown** interface for COM aggregation. Must be NULL.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_BADFORMAT
DSERR_GENERIC
DSERR_NODRIVER
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

Remarks

On Microsoft® Windows® 95, Windows® 98, and Windows® 2000, each capture device supports a single buffer.

The **IDirectSoundCaptureBuffer8** interface is supported only on buffers created by an object of class CLSID_DirectSoundCapture8. If the **IDirectSoundCapture8** interface was obtained from **DirectSoundCaptureCreate8**, **IDirectSoundCaptureBuffer8** is supported. If **IDirectSoundCapture8** was obtained

from the earlier **DirectSoundCaptureCreate** function, only **IDirectSoundCaptureBuffer** is supported.

Requirements

Header: Declared in dsound.h.

IDirectSoundCapture8::GetCaps

Obtains the capabilities of the capture device.

```
HRESULT GetCaps(
  LPDSCCAPS pDSCCaps
);
```

Parameters

pDSCCaps

Pointer to a **DSCCAPS** structure to be receive information about the capture device. When the method is called, the **dwSize** member must specify the size of the structure in bytes.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_UNSUPPORTED
DSERR_NODRIVER
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

Requirements

Header: Declared in dsound.h.

IDirectSoundCapture8::Initialize

When **CoCreateInstance** is used to create a DirectSoundCapture object, the object must be initialized with the **IDirectSoundCapture8::Initialize** method. Calling this method is not required when the **DirectSoundCaptureCreate** function is used to create the object.

```
HRESULT Initialize(
  LPCGUID pcGuidDevice
);
```

Parameters

pcGuidDevice
Address of the GUID specifying the sound driver to which the DirectSoundCapture object binds. Use NULL to select the primary sound driver.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_NODRIVER
DSERR_OUTOFMEMORY
DSERR_ALREADYINITIALIZED

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8

Used to manipulate sound capture buffers.

To obtain the **IDirectSoundCaptureBuffer8** interface, call the **IDirectSoundCapture8::CreateCaptureBuffer** method to obtain **IDirectSoundCaptureBuffer**, then pass IID_IDirectSoundCaptureBuffer8 to **IDirectSoundCaptureBuffer::QueryInterface**.

Note

IDirectSoundCaptureBuffer8 cannot be obtained if the **IDirectSoundCapture8** interface was obtained by using the **DirectSoundCaptureCreate** function. See **DirectSoundCaptureCreate8**.

The methods of this interface can be grouped as follows:

Capture management	**Lock**
	Start
	Stop
	Unlock
Effects	**GetFXStatus**
	GetObjectInPath
Initialization	**Initialize**
Information	**GetCaps**
	GetCurrentPosition

GetFormat
GetStatus

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

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTSOUNDCAPTUREBUFFER8** type is defined as a pointer to the **IDirectSoundCaptureBuffer8** interface:

```
typedef struct IDirectSoundCaptureBuffer8
*LPDIRECTSOUNDCAPTUREBUFFER8;
```

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::GetCaps

Returns the capabilities of the buffer.

```
HRESULT GetCaps(
  LPDSCBCAPS pDSCBCaps
);
```

Parameters

pDSCBCaps
Pointer to a **DSCBCAPS** structure that receives information about the capture buffer. On input, the **dwSize** member must specify the size of the structure in bytes.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_UNSUPPORTED
DSERR_OUTOFMEMORY

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::GetCurrentPosition

Retrieves the capture and read cursors in the buffer.

The capture cursor is ahead of the read cursor. The data after the read position up to and including the capture position is not necessarily valid data.

```
HRESULT GetCurrentPosition(
  LPDWORD pdwCapturePosition,
  LPDWORD pdwReadPosition
);
```

Parameters

pdwCapturePosition
Address of a variable that receives the offset from the start of the buffer, in bytes, of the capture cursor. The parameter can be NULL if this value is not wanted.

pdwReadPosition
Address of a variable that receives the offset from the start of the buffer, in bytes, of the read cursor.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_NODRIVER
DSERR_OUTOFMEMORY

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::GetFormat

Retrieves the format of the capture buffer.

```
HRESULT GetFormat(
  LPWAVEFORMATEX pwfxFormat,
  DWORD dwSizeAllocated,
  LPDWORD pdwSizeWritten
);
```

Parameters

pwfxFormat
Address of a **WAVEFORMATEX** structure that receives a description of the sound data in the capture buffer. To retrieve the buffer size needed to contain the format description, specify NULL.; in this case, the **DWORD** pointed to by the *pdwSizeWritten* parameter receives the size of the structure needed to receive complete format information.

dwSizeAllocated
Size, in bytes, of the **WAVEFORMATEX** structure. DirectSoundCapture writes, at most, *dwSizeAllocated* bytes to the structure; if the structure requires more memory, it is truncated.

pdwSizeWritten
Address of a variable that receives the number of bytes written to the **WAVEFORMATEX** structure; or, if *pwfxFormat* is NULL, the size of the **WAVEFORMATEX** structure that would be necessary to receive the information. This parameter can be NULL if the value is not wanted.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::GetFXStatus

Retrieves the status of effects for the capture buffer.

```
HRESULT GetFXStatus(
  DWORD dwFXCount,
  LPDWORD pdwFXStatus
);
```

Parameters

dwFXCount
Value that specifies the number of elements in the *pdwFXStatus* array.

pdwFXStatus
Address of an array of **DWORD** variables that receive the status of each effect. This should contain the same number of elements as the array passed in the *lpDSCFXDesc* parameter of

IDirectSoundCapture8::CreateCaptureBuffer.For each effect, one of the following flags is returned.

DSCFXR_LOCHARDWARE
Effect is instantiated in hardware.

DSCFXR_LOCSOFTWARE
Effect is instantiated in software.

DSCFXR_UNALLOCATED
Effect has not yet been assigned to hardware nor software.

DSCFXR_FAILED
No effect was created because resources weren't available.

DSCFXR_UNKNOWN
No effect was created because the effect isn't registered on the system.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::GetObjectInPath

Retrieves an interface for an effect object associated with the buffer.

```
HRESULT GetObjectInPath(
  REFGUID rguidObject,
  DWORD dwIndex,
  REFGUID rguidInterface,
  LPVOID * ppObject
);
```

Parameters

rguidObject
Value of type **REFGUID** that specifies the unique class identifier of the object being searched for, such as GUID_DSCFX_CLASS_AEC. Set this parameter to GUID_All_Objects to search for objects of any class.

dwIndex
Value that specifies the index of the object within objects of that class in the path. See Remarks.

rguidInterface

Value that specifies the unique identifier of the desired interface, such as IID_IDirectSoundCaptureFXAec8.

ppObject
Address of a variable that receives the desired interface pointer.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
E_NOINTERFACE

Remarks

The value in *dwIndex* is the index of the object within the array of effects passed to **IDirectSoundCapture8::CreateCaptureBuffer**. This is not necessarily the actual position of the object in the effects chain, because some effects might not have been created.

An object is returned solely on the basis of whether it matches *rguidObject* and *dwIndex*. It is up to the application to ensure that *rguidInterface* specifies an interface that can be expected to be found on the object.

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::GetStatus

Retrieves the status of the capture buffer.

```
HRESULT GetStatus(
  LPDWORD pdwStatus
);
```

Parameters

pdwStatus
Address of a variable that receives the status of the capture buffer. The status can be set to one or more of the following:

DSCBSTATUS_CAPTURING
The buffer is currently capturing audio data.

DSCBSTATUS_LOOPING
The buffer is currently looping.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::Initialize

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

```
HRESULT Initialize(
  LPDIRECTSOUNDCAPTURE pDirectSoundCapture,
  LPCDSCBUFFERDESC pcDSCBufferDesc
);
```

Parameters

pDirectSoundCapture
Address of the DirectSoundCapture object associated with this DirectSoundCaptureBuffer object.

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

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_ALREADYINITIALIZED

Remarks

Because the **IDirectSoundCapture8::CreateCaptureBuffer** method calls the **IDirectSoundCaptureBuffer8::Initialize** method internally, it is not needed for the current release of DirectSound. This method is provided for future extensibility.

Requirements

Header: Declared in dsound.h.

See Also

DSBUFFERDESC, **IDirectSoundCapture8::CreateCaptureBuffer**

IDirectSoundCaptureBuffer8::Lock

Locks the buffer. Locking the buffer returns pointers into the buffer, allowing the application to read or write audio data into that memory.

```
HRESULT Lock(
  DWORD dwOffset,
  DWORD dwBytes,
  LPVOID *ppvAudioPtr1,
  LPDWORD pdwAudioBytes1,
  LPVOID *ppvAudioPtr2,
  LPDWORD pdwAudioBytes2,
  DWORD dwFlags
);
```

Parameters

dwOffset
Value that specifies the offset, in bytes, from the start of the buffer to the point where the lock begins.

dwBytes
Value that specifies the size, in bytes, of the portion of the buffer to lock. Because the buffer is conceptually circular, this number can exceed the number of bytes between *dwOffset* and the end of the buffer.

ppvAudioPtr1
Address of a variable that receives a pointer to the first locked part of the buffer.

pdwAudioBytes1
Address of a variable that receives the number of bytes in the block at *ppvAudioPtr1*. If this value is less than *dwBytes*, the lock has wrapped and *ppvAudioPtr2* points to a second block of data at the beginning of the buffer.

ppvAudioPtr2
Address of a variable that receives a pointer to the second locked part of the capture buffer. If NULL is returned, the *ppvAudioPtr1* parameter points to the entire locked portion of the capture buffer.

pdwAudioBytes2
Address of a variable that receives the number of bytes in the block at *ppvAudioPtr2*. If *ppvAudioPtr2* is NULL, this value is zero.

dwFlags
Flags modifying the lock event. This value can be zero or the following flag:

DSCBLOCK_ENTIREBUFFER
Ignore *dwReadBytes* and lock the entire capture buffer.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_INVALIDCALL

Remarks

This method accepts an offset and a byte count, and returns two read pointers and their associated sizes. If the locked portion does not extend to the end of the buffer and wrap to the beginning, the second pointer, *ppvAudioBytes2*, receives NULL. If the lock does wrap, *ppvAudioBytes2* points to the beginning of the buffer.

If the application passes NULL for the *ppvAudioPtr2* and *pdwAudioBytes2* parameters, the lock extends no further than the end of the buffer and does not wrap.

The application should read data from the pointers returned by this method and then immediately call **IDirectSoundCaptureBuffer8::Unlock**. The sound buffer should not remain locked while it is running; if it does, the capture cursor will reach the locked bytes and audio problems may result.

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::Start

Begins capturing data into the buffer. If the buffer is already capturing, the method has no effect.

```
HRESULT Start(
  DWORD dwFlags
);
```

Parameters

dwFlags

Flags that specify the behavior of the buffer when capturing sound data. The following flag is defined:

DSCBSTART_LOOPING

After the end of the buffer is reached, capture restarts at the beginning and continues until explicitly stopped.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_NODRIVER
DSERR_OUTOFMEMORY

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::Stop

Stops the buffer so that it is no longer capturing data. If the buffer is not capturing, the method has no effect.

```
HRESULT Stop();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_NODRIVER
DSERR_OUTOFMEMORY

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureBuffer8::Unlock

Unlocks the buffer.

```
HRESULT Unlock(
  LPVOID pvAudioPtr1,
  DWORD dwAudioBytes1,
  LPVOID pvAudioPtr2,
  DWORD dwAudioBytes2
);
```

Parameters

pvAudioPtr1

Address of the value retrieved in the *ppvAudioPtr1* parameter of the **IDirectSoundCaptureBuffer8::Lock** method.

dwAudioBytes1
Number of bytes read from the *pvAudioPtr1* parameter. See Remarks.

pvAudioPtr2
Address of the value retrieved in the *ppvAudioPtr2* parameter of the **IDirectSoundCaptureBuffer8::Lock** method.

dwAudioBytes2
Number of bytes read from the *pvAudioPtr2* parameter. See Remarks.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_INVALIDCALL

Remarks

An application must pass both pointers, *pvAudioPtr1* and *pvAudioPtr2*, returned by the **IDirectSoundCaptureBuffer8::Lock** method to ensure the correct pairing of **IDirectSoundCaptureBuffer8::Lock** and **IDirectSoundCaptureBuffer8::Unlock**. The second pointer is needed even if zero bytes were written to the second pointer.

The values in *dwAudioBytes1* and *dwAudioBytes2* must specify the number of bytes actually read from each part of the buffer, which might be less than the size of the lock. DirectSound uses these values to determine how much data to transfer from the device.

Make sure that the capture buffer does not remain locked for long periods of time.

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureFXAec8

Not implemented on current operating systems.

Used to set and retrieve parameters on a capture buffer that supports acoustic echo cancellation.

IDirectSoundCaptureFXAec8 is a define for **IDirectSoundCaptureFXAec**. The interface names are interchangeable.

This interface is obtained by calling **IDirectSoundCaptureBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundCaptureFXAec8** interface has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

Like all COM interfaces, the **IDirectSoundCaptureFXAec8** interface inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTSOUNDCAPTUREFXAEC8** type is defined as a pointer to the **IDirectSoundCaptureFXAec8** interface:

```
typedef struct IDirectSoundCaptureFXAec8
*LPDIRECTSOUNDCAPTUREFXAEC8;
```

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureFXAec8::GetAllParameters

Not implemented on current operating systems.

Retrieves the acoustic echo cancellation parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSCFXAec pDscFxAec
);
```

Parameters

pDscFxAec
Address of a **DSCFXAec** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureFXAec8::SetAllParameters

Not implemented on current operating systems.

Sets the acoustic echo cancellation parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSCFXAec  pcDscFxAec
);
```

Parameters

pcDscFxAec
Address of a **DSCFXAec** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureFXNoiseSuppress8

Not implemented on current operating systems.

Used to set and retrieve parameters on a capture buffer that supports noise suppression.

IDirectSoundCaptureFXNoiseSuppress8 is a define for **IDirectSoundCaptureFXNoiseSuppress**. The interface names are interchangeable.

This interface is obtained by calling **IDirectSoundCaptureBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundCaptureFXNoiseSuppress8** interface has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

Like all COM interfaces, the **IDirectSoundNoiseSuppress8** interface inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTSOUNDNOISESUPPRESS8** type is defined as a pointer to the **IDirectSoundNoiseSuppress8** interface:

```
typedef struct IDirectSoundCaptureFXNoiseSuppress8
*LPDIRECTSOUNDCAPTUREFXNOISESUPPRESS8;
```

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureFXNoiseSuppress8::GetAllParameters

Not implemented on current operating systems.

Retrieves the noise suppression parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSCFXNoiseSuppress pDsFxNoiseSuppress
);
```

Parameters

pDscFxNoiseSuppress
Address of a **DSCFXNoiseSuppress** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundCaptureFXNoiseSuppress8::SetAllParameters

Not implemented on current operating systems.

Sets the noise suppression parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSCFXNoiseSuppression  pcDscFxNoiseSuppress
);
```

Parameters

pcDscFxNoiseSuppress
Address of a **DSCFXNoiseSuppress** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFullDuplex8

Not implemented on current operating systems.

Represents a full-duplex stream.

The interface is obtained by using the **DirectSoundFullDuplexCreate8** function. This function initializes DirectSound capture and playback.

The interface can also be obtained by using **CoCreateInstance** or **CoCreateInstanceEx** to create an object of class CLSID_DirectSoundFullDuplex8.

IDirectSoundFullDuplex8 is a define for **IDirectSoundFullDuplex**. The interface names are interchangeable.

This interface has the following method:

Initialization	**Initialize**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**

QueryInterface
Release

Requirements

Header: Declared in dsound.h.

The **LPDIRECTSOUNDFULLDUPLEX** type is defined as a pointer to **IDirectSoundFullDuplex**.

```
typedef struct IDirectSoundFullDuplex *LPDIRECTSOUNDFULLDUPLEX;
```

IDirectSoundFullDuplex8::Initialize

Initializes a full duplex object that was created by using **CoCreateInstance**.

```
HRESULT Initialize(
  LPCGUID  pCaptureGuid,
  LPCGUID  pRenderGuid,
  LPCDSCBUFFERDESC  lpDscBufferDesc,
  LPCDSBUFFERDESC  lpDsBufferDesc,
  HWND  hWnd,
  DWORD  dwLevel,
  LPLPDIRECTSOUNDCAPTUREBUFFER8  lplpDirectSoundCaptureBuffer8,
  LPLPDIRECTSOUNDBUFFER8  lplpDirectSoundBuffer8
);
```

Parameters

pCaptureGuid
Address of the GUID that identifies the sound capture device for full duplex input. Must be one of the GUIDs returned by **DirectSoundCaptureEnumerate**, NULL for the default capture device, or one of the following values:

DSDEVID_DefaultCapture
System-wide default audio capture device.

DSDEVID_DefaultVoiceCapture
Default voice capture device.

pRenderGuid
Address of the GUID that identifies the sound render device for full-duplex output. Must be one of the GUIDs returned by **DirectSoundEnumerate**, NULL for the default render device, or one of the following values:

DSDEVID_DefaultPlayback
System-wide default audio playback device.

DSDEVID_DefaultVoicePlayback
Default voice playback device.

lpDscBufferDesc
Address of a **DSCBUFFERDESC** structure that specifies the characteristics of the capture buffer.

lpDsBufferDesc
Address of a **DSBUFFERDESC** structure that specifies the characteristics of the render buffer.

hWnd
Handle to the application window.

dwLevel
Cooperative level for the DirectSound object. For more information, see **IDirectSound8::SetCooperativeLevel**.

lplpDirectSoundCaptureBuffer8
Address of a variable that receives the **IDirectSoundCaptureBuffer8** interface of the capture buffer.

lplpDirectSoundBuffer8
Address of a variable that receives the **IDirectSoundBuffer8** interface of the render buffer.

Requirements

Header: Declared in dsound.h.

IDirectSoundFXChorus8

Used to set and retrieve effect parameters on a buffer that supports chorus.

This interface is obtained by calling **IDirectSoundBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundFXChorus8** interface has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX

IDirectSoundFXChorus8::GetAllParameters

Retrieves the chorus parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSFXChorus pDsFxChorus
);
```

Parameters

pDsFxChorus
Address of a **DSFXChorus** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFXChorus8::SetAllParameters

Sets the chorus parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSFXChorus  pcDsFxChorus
);
```

Parameters

pcDsFxChorus
Address of a **DSFXChorus** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundFXCompressor8

Used to set and retrieve effect parameters on a buffer that supports compression.

This interface is obtained by calling **IDirectSoundBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundFXCompressor8** interface has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX, Compression

IDirectSoundFXCompressor8::GetAllParameters

Retrieves the compression parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSFXCompressor pDsFxCompressor
);
```

Parameters

pDsFxCompressor
Address of a **DSFXCompressor** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFXCompressor8::SetAllParameters

Sets the compression parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSFXCompressor pcDsFxCompressor
);
```

Parameters

pcDsFxCompressor
Address of a **DSFXCompressor** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundFXDistortion8

Used to set and retrieve effect parameters on a buffer that supports distortion.

This interface is obtained by calling **IDirectSoundBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundFXDistortion8** interface has the following methods.

Parameters **GetAllParameters**

SetAllParameters

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX

IDirectSoundFXDistortion8::GetAllParameters

Retrieves the distortion parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSFXDistortion  pDsFxDistortion
);
```

Parameters

pDsFxDistortion
Address of a **DSFXDistortion** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFXDistortion8::SetAllParameters

Sets the distortion parameters of a buffer.

```
HRESULT SetAllParameters(
```

LPCDSFXDistortion *pcDsFxDistortion*
);

Parameters

pcDsFxDistortion
Address of a **DSFXDistortion** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundFXEcho8

Used to set and retrieve effect parameters on a buffer that supports echo.

This interface is obtained by calling **IDirectSoundBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundFXEcho8** interface has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX

IDirectSoundFXEcho8::GetAllParameters

Retrieves the echo parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSFXEcho pDsFxEcho
);
```

Parameters

pDsFxEcho
Address of a **DSFXEcho** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFXEcho8::SetAllParameters

Sets the echo parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSFXEcho  pcDsFxEcho
);
```

Parameters

pcDsFxEcho
Address of a **DSFXEcho** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundFXFlanger8

Used to set and retrieve effect parameters on a buffer that supports flange.

This interface is obtained by calling **IDirectSoundBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundFXFlanger8** interface has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX

IDirectSoundFXFlanger8::GetAllParameters

Retrieves the flange parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSFXFlanger  pDsFxFlanger
);
```

Parameters

pDsFxFlanger
Address of a **DSFXFlanger** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFXFlanger8::SetAllParameters

Sets the flange parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSFXFlanger pcDsFxFlanger
);
```

Parameters

pcDsFxFlanger
Address of a **DSFXFlanger** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundFXGargle8

Used to set and retrieve effect parameters on a buffer that supports amplitude modulation.

This interface is obtained by calling **IDirectSoundBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundFXGargle8** interface has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**

QueryInterface
Release

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX

IDirectSoundFXGargle8::GetAllParameters

Retrieves the amplitude modulation parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSFXGargle  pDsFxGargle
);
```

Parameters

pDsFxGargle
Address of a **DSFXGargle** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFXGargle8::SetAllParameters

Sets the amplitude modulation parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSFXGargle  pcDsFxGargle
);
```

Parameters

pcDsFxGargle

Address of a **DSFXGargle** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundFXI3DL2Reverb8

Used to set and retrieve effect parameters on a buffer that supports I3DL2 (Interactive 3D Audio Level 2) reverberation effects.

This interface is obtained by calling **IDirectSoundBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundFXI3DL2Reverb8** interface has the following methods.

Parameters	**GetAllParameters**
	GetQuality
	SetAllParameters
	SetQuality
Presets	**GetPreset**
	SetPreset

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX

IDirectSoundFXI3DL2Reverb8::GetAllParameters

Retrieves the I3DL2 environmental reverberation parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSFXI3DL2Reverb pDsFxI3DL2Reverb
);
```

Parameters

pDsFxI3DL2Reverb
Address of a **DSFXI3DL2Reverb** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFXI3DL2Reverb8::GetPreset

Retrieves an identifier for standard reverberation parameters of a buffer.

```
HRESULT GetPreset(
  LPDWORD  pdwPreset
);
```

Parameters

pdwPreset

Address of a DWORD that receives one of the preset identifiers. For a list of defined values, see IDirectSoundFXI3DL2Reverb8::SetPreset.Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_INVALIDCALL

Requirements

Header: Declared in dsound.h.

IDirectSoundFXI3DL2Reverb8::GetQuality

Retrieves the quality of the environmental reverberation effect.

```
HRESULT GetQuality(
  LONG * plQuality
);
```

Parameters

plQuality
Address of a **LONG** variable to receive the quality of the effect.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXI3DL2Reverb8::SetQuality

IDirectSoundFXI3DL2Reverb8::SetAllParameters

Sets the I3DL2 environmental reverberation parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSFXI3DL2Reverb  pcDsFxI3DL2Reverb
);
```

Parameters

pcDsFxI3DL2Reverb
Address of a **DSFXI3DL2Reverb** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundFXI3DL2Reverb8::SetPreset

Specifies standard reverberation parameters of a buffer.

```
HRESULT SetPreset(
  DWORD  dwPreset
);
```

Parameters

dwPreset

Value that specifies a preset. The following constants are enumerated. See Dsound.h for the default values associated with each preset.

DSFX_I3DL2_ENVIRONMENT_PRESET_DEFAULT
DSFX_I3DL2_ENVIRONMENT_PRESET_GENERIC
DSFX_I3DL2_ENVIRONMENT_PRESET_PADDEDCELL
DSFX_I3DL2_ENVIRONMENT_PRESET_ROOM
DSFX_I3DL2_ENVIRONMENT_PRESET_BATHROOM
DSFX_I3DL2_ENVIRONMENT_PRESET_LIVINGROOM
DSFX_I3DL2_ENVIRONMENT_PRESET_STONEROOM
DSFX_I3DL2_ENVIRONMENT_PRESET_AUDITORIUM
DSFX_I3DL2_ENVIRONMENT_PRESET_CONCERTHALL
DSFX_I3DL2_ENVIRONMENT_PRESET_CAVE
DSFX_I3DL2_ENVIRONMENT_PRESET_ARENA
DSFX_I3DL2_ENVIRONMENT_PRESET_HANGAR
DSFX_I3DL2_ENVIRONMENT_PRESET_CARPETEDHALLWAY
DSFX_I3DL2_ENVIRONMENT_PRESET_HALLWAY
DSFX_I3DL2_ENVIRONMENT_PRESET_STONECORRIDOR
DSFX_I3DL2_ENVIRONMENT_PRESET_ALLEY
DSFX_I3DL2_ENVIRONMENT_PRESET_FOREST

DSFX_I3DL2_ENVIRONMENT_PRESET_CITY
DSFX_I3DL2_ENVIRONMENT_PRESET_MOUNTAINS
DSFX_I3DL2_ENVIRONMENT_PRESET_QUARRY
DSFX_I3DL2_ENVIRONMENT_PRESET_PLAIN
DSFX_I3DL2_ENVIRONMENT_PRESET_PARKINGLOT
DSFX_I3DL2_ENVIRONMENT_PRESET_SEWERPIPE
DSFX_I3DL2_ENVIRONMENT_PRESET_UNDERWATER
DSFX_I3DL2_ENVIRONMENT_PRESET_SMALLROOM
DSFX_I3DL2_ENVIRONMENT_PRESET_MEDIUMROOM
DSFX_I3DL2_ENVIRONMENT_PRESET_LARGEROOM
DSFX_I3DL2_ENVIRONMENT_PRESET_MEDIUMHALL
DSFX_I3DL2_ENVIRONMENT_PRESET_LARGEHALL
DSFX_I3DL2_ENVIRONMENT_PRESET_PLATE

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundFXI3DL2Reverb8::SetQuality

Sets the quality of the environmental reverberation effect. Higher values produce better quality at the expense of processing time.

```
HRESULT SetQuality(
  LONG lQuality
);
```

Parameters

lQuality

Value that specifies the quality of the effect, in the range from DSFX_I3DL2REVERB_QUALITY_MIN to DSFX_I3DL2REVERB_QUALITY_MAX. The default value is 2.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXI3DL2Reverb8::GetQuality

See Also

Effect Parameters

IDirectSoundFXParamEq8

Used to set and retrieve effect parameters on a buffer that supports parametric equalizer effects.

The **IDirectSoundFXParamEq8** interface has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX

IDirectSoundFXParamEq8::GetAllParameters

Retrieves the parametric equalizer parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSFXParamEq pDsFxParamEq
);
```

Parameters

pDsFxParamEq
Address of a **DSFXParamEq** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFXParamEq8::SetAllParameters

Sets the parametric equalizer parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSFXParamEq  pcDsFxParamEq
);
```

Parameters

pcDsFxParamEq
Address of a **DSFXParamEq** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundFXWavesReverb8

Used to set and retrieve effect parameters on a buffer that supports music reverberation.

This interface is obtained by calling **IDirectSoundBuffer8::GetObjectInPath** on the buffer that supports the effect.

The **IDirectSoundFXWavesReverb8** interface has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

Remarks

The Waves reverberation DMO is based on the Waves MaxxVerb technology, which is licenced to Microsoft.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundBuffer8::SetFX

IDirectSoundFXWavesReverb8::GetAllParameters

Retrieves the music reverberation parameters of a buffer.

```
HRESULT GetAllParameters(
  LPDSFXWavesReverb  pDsFXWavesReverb
);
```

Parameters

pDsFXWavesReverb
Address of a **DSFXWavesReverb** structure that receives the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

IDirectSoundFXWavesReverb8::SetAllParameters

Sets the music reverberation parameters of a buffer.

```
HRESULT SetAllParameters(
  LPCDSFXWavesReverb  pcDsFXWavesReverb
);
```

Parameters

pcDsFXWavesReverb
Address of a **DSFXWavesReverb** structure that specifies the effect parameters.

Return Values

If the method succeeds, the return value is DS_OK.

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

Requirements

Header: Declared in dsound.h.

See Also

Effect Parameters

IDirectSoundNotify8

Sets up notification events for a playback or capture buffer.

IDirectSoundNotify8 is a define for **IDirectSoundNotify**. The two interface names are interchangeable.

The interface is obtained by calling the **QueryInterface** method of an existing interface on a DirectSoundBuffer object. For an example, see Play Buffer Notification.

The interface has the following method:

SetNotificationPositions

Like all COM interfaces, the **IDirectSoundNotify8** interface inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECTSOUNDNOTIFY** type is defined as a pointer to the **IDirectSoundNotify** interface:

```
typedef struct IDirectSoundNotify    *LPDIRECTSOUNDNOTIFY;
```

Requirements

Header: Declared in dsound.h.

IDirectSoundNotify8::SetNotificationPositions

Sets the notification positions. During capture or playback, whenever the read or play cursor reaches an offset specified in one of the **DSBPOSITIONNOTIFY** structures in the caller-supplied array, the associated event is signaled.

```
HRESULT SetNotificationPositions(
  DWORD dwPositionNotifies,
  LPCDSBPOSITIONNOTIFY pcPositionNotifies
);
```

Parameters

dwPositionNotifies
Number of **DSBPOSITIONNOTIFY** structures.

pcPositionNotifies
Pointer to an array of **DSBPOSITIONNOTIFY** structures.

Return Values

If the method succeeds, the return value is DS_OK.

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

DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY

Remarks

The value DSBPN_OFFSETSTOP can be specified in the **dwOffset** member to tell DirectSound to signal the associated event when the **IDirectSoundBuffer8::Stop** or **IDirectSoundCaptureBuffer8::Stop** method is called or when the end of the buffer has been reached and the playback is not looping. If it is used, this should be the last item in the position-notify array.

If a position-notify array has already been set, the method replaces the previous array.

The buffer must be stopped when this method is called.

Requirements

Header: Declared in dsound.h.

IKsPropertySet

Allows drivers to provide extended capabilities that can be used without API extensions. DirectSound supports this interface for accessing properties of sound drivers.

The **IKsPropertySet** interface is part of the WDM KS (Win32® driver model kernel streaming) architecture.

To retrieve this interface, call the **QueryInterface** method of any interface on a DirectSound buffer.

The **IKsPropertySet** interface has the following methods:

IKsPropertySet	**Get**
	QuerySupport
	Set

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in dsound.h.

IKsPropertySet::Get

Retrieves data for an item in a property set.

```
HRESULT Get(
  REFGUID rguidPropSet,
  ULONG ulId,
  LPVOID pInstanceData,
  ULONG ulInstanceLength,
  LPVOID pPropertyData,
  ULONG ulDataLength,
  PULONG pulBytesReturned
);
```

Parameters

rguidPropSet
Reference to (C++) or address of (C) a GUID representing the property set to be accessed.

ulId
Item within the property set to be accessed. Items are indexed from 0 and are always the same for a given property set GUID.

pInstanceData
Instance data for the get call.

ulInstanceLength
Number of bytes pointed to by *pInstanceData.*

pPropertyData
Data to set for this item.

ulDataLength
Number of bytes pointed to by *pPropertyData*

pulBytesReturned
Address of a variable that receives the number of bytes written into *pPropertyData*.

Return Values

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

If the method succeeds, the return value may be S_OK.

If it fails, the method may return one E_POINTER.

Remarks

The format of the data in both *pInstanceData* and *pPropertyData* is item-specific.

Requirements

Header: Declared in dsound.h.

IKsPropertySet::QuerySupport

Determines whether a property in a property set is supported on the port or device.

```
HRESULT QuerySupport(
  REFGUID rguidPropSet,
  ULONG ulId,
  PULONG pulTypeSupport
);
```

Parameters

rguidPropSet
Reference to (C++) or address of (C) a GUID representing the property set to be queried.

ulId
Item within the property set to be accessed. Items are indexed from 0 and are always the same for a given property set.

pulTypeSupport
Address of a variable that receives information about support for the property. This variable can receive one or both of the following flags. See Remarks:

KSPROPERTY_SUPPORT_GET
The property item may be retrieved.

KSPROPERTY_SUPPORT_SET
The property item may be set.

Return Values

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

If the method succeeds, the return value may be S_OK. (See Remarks.)

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

E_NOTIMPL (See Remarks.)

E_POINTER

Remarks

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

Some implementations may return S_OK when the property is not supported, and others may return E_NOTIMPL. To be sure that a property is supported, check both the return value of the method and the value returned in *pulTypeSupport*. If S_OK is returned but *pulTypeSupport* does not contain a valid flag, the property is not supported.

Requirements

Header: Declared in dsound.h.

IKsPropertySet::Set

Sets the value of a property in a property set.

```
HRESULT Set(
  REFGUID rguidPropSet,
  ULONG ulId,
  LPVOID pInstanceData,
  ULONG ulInstanceLength,
  LPVOID pPropertyData,
  ULONG ulDataLength
);
```

Parameters

rguidPropSet
Reference to (C++) or address of (C) a GUID representing the property set to be accessed.

ulId
Item within the property set to be accessed. Items are indexed from 0 and are always the same for a given property set.

pInstanceData
Instance data for the set call. If there are multiple objects within the port that this operation could act on, the instance data specifies which object should be used. No standard property set items at the DirectMusic® API level use instance data; however, vendor-defined extensions are free to use it.

ulInstanceLength
Number of bytes pointed to by *pInstanceData*.

pPropertyData
Property data to set for this item.

ulDataLength
Number of bytes pointed to by *pPropertyData*.

Return Values

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

If the method succeeds, the return value may be S_OK.

If it fails, the method may return E_POINTER.

Remarks

The format of the data in both *pInstanceData* and *pPropertyData* is item-specific.

Requirements

Header: Declared in dsound.h.

DirectSound Callback Function

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

- **DSEnumCallback**

DSEnumCallback

Application-defined callback function that enumerates the DirectSound drivers. The system calls this function in response to the application's previous call to the **DirectSoundEnumerate** or **DirectSoundCaptureEnumerate** function.

```
BOOL CALLBACK DSEnumCallback(
  LPGUID lpGuid,
  LPCSTR lpcstrDescription,
  LPCSTR lpcstrModule,
  LPVOID lpContext
);
```

Parameters

lpGuid

Address of the GUID that identifies the DirectSound driver being enumerated. This value can be passed to the **DirectSoundCreate8** function to create a DirectSound object for that driver.

lpcstrDescription

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

lpcstrModule

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

lpContext

Address of application-defined data. This is the pointer passed to **DirectSoundEnumerate** or **DirectSoundCaptureEnumerate** as the *lpContext* parameter.

Return Values

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

Remarks

The application can save the strings passed in the *lpcstrDescription* and *lpcstrModule* parameters by copying them to memory allocated from the heap. The memory used to pass the strings to this callback function is valid only while this callback function is running.

Requirements

Header: Declared in dsound.h.

See Also

DirectSoundEnumerate

DirectSound Functions

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

- **DirectSoundCaptureCreate8**
- **DirectSoundFullDuplexCreate8**
- **DirectSoundCaptureEnumerate**
- **DirectSoundCreate8**
- **DirectSoundEnumerate**
- **GetDeviceID**

DirectSoundCaptureCreate8

Creates and initializes an object that supports the **IDirectSoundCapture8** interface.

This function creates an object that supports capture effects. Although the older **DirectSoundCaptureCreate** function can also be used to obtain the **IDirectSoundCapture8** interface, the object created by that function cannot be used to create capture buffers that support the **IDirectSoundCaptureBuffer8** interface.

HRESULT WINAPI DirectSoundCaptureCreate8(
LPCGUID *lpcGUID*,

```
  LPDIRECTSOUNDCAPTURE8 *lplpDSC,
  LPUNKNOWN pUnkOuter
);
```

Parameters

lpcGUID
Address of the GUID that identifies the sound capture device. The value of this parameter must be one of the GUIDs returned by **DirectSoundCaptureEnumerate**, NULL for the default device, or one of the following values:

DSDEVID_DefaultCapture
System-wide default audio capture device.

DSDEVID_DefaultVoiceCapture
Default voice capture device.

lplpDSC
Address of a variable to receive an **IDirectSoundCapture8** interface pointer.

pUnkOuter
Address of the controlling object's **IUnknown** interface for COM aggregation. Must be NULL, because aggregation is not supported.

Return Values

If the function succeeds, the return value is DS_OK.

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

DSERR_ALLOCATED
DSERR_INVALIDPARAM
DSERR_NOAGGREGATION
DSERR_OUTOFMEMORY

Remarks

On sound cards that do not support full duplex, this method will fail and return DSERR_ALLOCATED.

Requirements

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

DirectSoundCaptureEnumerate

Enumerates the DirectSoundCapture objects installed in the system.

```
HRESULT WINAPI DirectSoundCaptureEnumerate(
  LPDSENUMCALLBACK lpDSEnumCallback,
  LPVOID lpContext
);
```

Parameters

lpDSEnumCallback
Address of the **DSEnumCallback** function that will be called for each DirectSoundCapture object installed in the system.

lpContext
Address of the user-defined context passed to the enumeration callback function every time that function is called.

Return Values

If the function succeeds, the return value is DS_OK.

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

Requirements

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

DirectSoundCreate8

Creates and initializes an object that supports the **IDirectSound8** interface.

```
HRESULT WINAPI DirectSoundCreate8(
  LPCGUID lpcGuidDevice,
  LPDIRECTSOUND8 * ppDS8,
  LPUNKNOWN  pUnkOuter
);
```

Parameters

lpcGuidDevice
Address of the GUID that identifies the sound device. The value of this parameter must be one of the GUIDs returned by **DirectSoundEnumerate**, NULL for the default device, or one of the following values:

DSDEVID_DefaultPlayback
System-wide default audio playback device. Equivalent to NULL.

DSDEVID_DefaultVoicePlayback
Default voice playback device.

ppDS8
Address of a variable to receive an **IDirectSound8** interface pointer.

pUnkOuter
Address of the controlling object's **IUnknown** interface for COM aggregation. Must be NULL, because aggregation is not supported.

Return Values

If the function succeeds, the return value is DS_OK.

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

DSERR_ALLOCATED
DSERR_INVALIDPARAM
DSERR_NOAGGREGATION
DSERR_NODRIVER
DSERR_OUTOFMEMORY

Remarks

The application must call the **IDirectSound8::SetCooperativeLevel** method immediately after creating a DirectSound object.

Requirements

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

See Also

IDirectSound8::GetCaps, **IDirectSound8::SetCooperativeLevel**

DirectSoundEnumerate

Enumerates the DirectSound drivers installed in the system.

```
HRESULT WINAPI DirectSoundEnumerate(
  LPDSENUMCALLBACK lpDSEnumCallback,
  LPVOID lpContext
);
```

Parameters

lpDSEnumCallback
Address of the **DSEnumCallback** function that will be called for each DirectSound object installed in the system.

lpContext
Address of the user-defined context passed to the enumeration callback function every time that function is called.

Return Values

If the function succeeds, the return value is DS_OK.

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

Requirements

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

DirectSoundFullDuplexCreate8

Creates the DirectSound and DirectSoundCapture objects and returns the **IDirectSoundFullDuplex8** interface.

The **IDirectSoundFullDuplex8** object is not supported on current operating systems. However, this function can be used for convenience to set up the other objects necessary for full-duplex operations.

```
HRESULT WINAPI DirectSoundFullDuplexCreate8(
  LPCGUID pcGuidCaptureDevice,
  LPCGUID pcGuidRenderDevice,
  LPCDSCBUFFERDESC pcDSCBufferDesc,
  LPCDSBUFFERDESC pcDSBufferDesc,
  HWND hWnd,
  DWORD dwLevel,
  LPDIRECTSOUNDFULLDUPLEX* ppDSFD,
  LPDIRECTSOUNDCAPTUREBUFFER8 *ppDSCBuffer8,
  LPDIRECTSOUNDBUFFER8 *ppDSBuffer8,
  LPUNKNOWN pUnkOuter
);
```

Parameters

pcGuidCaptureDevice

Address of the GUID that identifies the sound capture device for full duplex input. Must be one of the GUIDs returned by **DirectSoundCaptureEnumerate**, NULL for the default capture device, or one of the following values:

DSDEVID_DefaultCapture
System-wide default audio capture device.

DSDEVID_DefaultVoiceCapture
Default voice capture device.

pcGuidRenderDevice

Address of the GUID that identifies the sound render device for full-duplex output. Must be one of the GUIDs returned by **DirectSoundEnumerate**, or NULL for the default render device, or one of the following values:

DSDEVID_DefaultPlayback
System-wide default audio playback device.

DSDEVID_DefaultVoicePlayback
Default voice playback device.

pcDSCBufferDesc
Address of a **DSCBUFFERDESC** structure that specifies the characteristics of the capture buffer.

pcDSBufferDesc
Address of a **DSBUFFERDESC** structure that specifies the characteristics of the render buffer.

hWnd
Handle to the application window.

dwLevel
Cooperative level for the DirectSound object. For more information, see **IDirectSound8::SetCooperativeLevel**.

ppDSFD
Address of a variable that receives the **IDirectSoundFullDuplex8** interface pointer.

ppDSCBuffer8
Address of a variable that receives the **IDirectSoundCaptureBuffer8** interface of the capture buffer.

ppDSBuffer8
Address of a variable that receives the **IDirectSoundBuffer8** interface of the render buffer.

pUnkOuter
Address of the controlling object's **IUnknown** interface for COM aggregation. Must be NULL, because aggregation is not supported.

Requirements

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

GetDeviceID

Retrieves the unique device identifier of the default playback and capture devices selected by the user under Sounds and Multimedia in Control Panel.

HRESULT GetDeviceID(
LPCGUID *pGuidSrc*,
LPGUID *pGuidDest*

);

Parameters

pGuidSrc

Address of a variable of type **GUID** that specifies a valid device identifier, or the address of one of the following predefined variables.

DSDEVID_DefaultPlayback
System-wide default audio playback device.

DSDEVID_DefaultCapture
System-wide default audio capture device.

DSDEVID_DefaultVoicePlayback
Default voice playback device.

DSDEVID_DefaultVoiceCapture
Default voice capture device.

pGuidDest

Address of a variable of type **GUID** that receives the unique identifier of the device.

Return Values

If the function succeeds, the return value is DS_OK.

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

Remarks

If *pGuidSrc* points to a valid device identifier, the same value is returned in *pGuidDest*. If pGuidSrc is one of the listed constants, *pGuidDest* returns the address of the corresponding device GUID.

Requirements

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

DirectSound Structures

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

- **DS3DBUFFER**
- **DS3DLISTENER**

- **DSBCAPS**
- **DSBPOSITIONNOTIFY**
- **DSBUFFERDESC**
- **DSCAPS**
- **DSCBCAPS**
- **DSCBUFFERDESC**
- **DSCCAPS**
- **DSCEFFECTDESC**
- **DSCFXAec**
- **DSCFXNoiseSuppress**
- **DSEFFECTDESC**
- **DSFXI3DL2Reverb**
- **DSFXChorus**
- **DSFXCompressor**
- **DSFXDistortion**
- **DSFXEcho**
- **DSFXFlanger**
- **DSFXGargle**
- **DSFXParamEq**
- **DSFXWavesReverb**
- **WAVEFORMATEX**
- **WAVEFORMATEXTENSIBLE**

Note

The memory for all DirectX structures must be initialized to zero before use. In addition, all structures that contain a **dwSize** member must set the member to the size of the structure, in bytes, before use. The following example performs these tasks on a common structure, **DSCAPS**:

```
DSCAPS dscaps; // Can't use this yet.

ZeroMemory(&dscaps, sizeof(dscaps));
dscaps.dwSize = sizeof(dscaps);

// Now the structure can be used.
.
.
.
```

DS3DBUFFER

Contains all information necessary to uniquely describe the location, orientation, and motion of a 3-D sound buffer. This structure is used with the **IDirectSound3DBuffer8::GetAllParameters** and **IDirectSound3DBuffer8::SetAllParameters** methods.

```
typedef struct {
    DWORD      dwSize;
    D3DVECTOR  vPosition;
    D3DVECTOR  vVelocity;
    DWORD      dwInsideConeAngle;
    DWORD      dwOutsideConeAngle;
    D3DVECTOR  vConeOrientation;
    LONG       lConeOutsideVolume;
    D3DVALUE   flMinDistance;
    D3DVALUE   flMaxDistance;
    DWORD      dwMode;
} DS3DBUFFER, *LPDS3DBUFFER;

typedef const DS3DBUFFER *LPCDS3DBUFFER;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

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

vVelocity
A **D3DVECTOR** structure that describes the current velocity of the 3-D sound buffer.

dwInsideConeAngle
The angle of the inside sound projection cone.

dwOutsideConeAngle
The angle of the outside sound projection cone.

vConeOrientation
A **D3DVECTOR** structure that describes the current orientation of this 3-D buffer's sound projection cone.

lConeOutsideVolume
The cone outside volume.

flMinDistance
The minimum distance.

flMaxDistance

The maximum distance.

dwMode

The 3-D sound processing mode to be set.

DS3DMODE_DISABLE
3-D sound processing is disabled. The sound will appear to originate from the center of the listener's head.

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

DS3DMODE_NORMAL
Normal processing. This is the default mode.

Requirements

Header: Declared in dsound.h.

DS3DLISTENER

Describes the 3-D world parameters and position of the listener. This structure is used with the **IDirectSound3DListener8::GetAllParameters** and **IDirectSound3DListener8::SetAllParameters** methods.

```
typedef struct {
    DWORD      dwSize;
    D3DVECTOR  vPosition;
    D3DVECTOR  vVelocity;
    D3DVECTOR  vOrientFront;
    D3DVECTOR  vOrientTop;
    D3DVALUE   flDistanceFactor;
    D3DVALUE   flRolloffFactor;
    D3DVALUE   flDopplerFactor;
} DS3DLISTENER, *LPDS3DLISTENER;

typedef const DS3DLISTENER *LPCDS3DLISTENER;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

vPosition, **vVelocity**, **vOrientFront**, **vOrientTop**

D3DVECTOR structures that describe the listener's position, velocity, front orientation, and top orientation, respectively.

flDistanceFactor, **flRolloffFactor**, **flDopplerFactor**

The current distance, rolloff, and Doppler factors, respectively.

Requirements

Header: Declared in dsound.h.

DSBCAPS

Describes the capabilities of a DirectSound buffer object. Used by the **IDirectSoundBuffer8::GetCaps** method.

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

typedef const DSBCAPS *LPCDSBCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Flags that specify buffer-object capabilities.

DSBCAPS_CTRL3D
The buffer has 3-D control capability.

DSBCAPS_CTRLFREQUENCY
The buffer has frequency control capability.

DSBCAPS_CTRLFX
The buffer supports effects processing.

DSBCAPS_CTRLPAN
The buffer has pan control capability.

DSBCAPS_CTRLVOLUME
The buffer has volume control capability.

DSBCAPS_CTRLPOSITIONNOTIFY
The buffer has position notification capability. See the Remarks for **DSBUFFERDESC**.

DSBCAPS_GETCURRENTPOSITION2
Indicates that **IDirectSoundBuffer8::GetCurrentPosition** should use the new behavior of the play cursor. In DirectSound in DirectX 1, the play cursor was significantly ahead of the actual playing sound on emulated sound cards;

it was directly behind the write cursor. Now, if the DSBCAPS_GETCURRENTPOSITION2 flag is specified, the application can get a more accurate play cursor. If this flag is not specified, the old behavior is preserved for compatibility. Note that this flag affects only emulated sound cards; if a DirectSound driver is present, the play cursor is accurate for DirectSound in all versions of DirectX.

DSBCAPS_GLOBALFOCUS
The buffer is a global sound buffer. With this flag set, an application using DirectSound can continue to play its buffers if the user switches focus to another application, even if the new application uses DirectSound. The one exception is if you switch focus to a DirectSound application that uses the DSSCL_WRITEPRIMARY flag for its cooperative level. In this case, the global sounds from other applications will not be audible.

DSBCAPS_LOCDEFER
The buffer can be assigned to a hardware or software resource at play time, or when **IDirectSoundBuffer8::AcquireResources** is called.

DSBCAPS_LOCHARDWARE
The buffer uses hardware mixing.

DSBCAPS_LOCSOFTWARE
The buffer is in software memory and uses software mixing.

DSBCAPS_MUTE3DATMAXDISTANCE
The sound is reduced to silence at the maximum distance. The buffer will stop playing when the maximum distance is exceeded, so that processor time is not wasted.

DSBCAPS_PRIMARYBUFFER
The buffer is a primary sound buffer.

DSBCAPS_STATIC
The buffer is in on-board hardware memory.

DSBCAPS_STICKYFOCUS
The buffer has sticky focus. If the user switches to another application not using DirectSound, the buffer is still audible. However, if the user switches to another DirectSound application, the buffer is muted.

dwBufferBytes
Size of this buffer, in bytes.

dwUnlockTransferRate
Specifies the rate, in kilobytes per second, at which data is transferred to the buffer memory when **IDirectSoundBuffer8::Unlock** is called. High-performance applications can use this value to determine the time required for **IDirectSoundBuffer8::Unlock** to execute. For software buffers located in system memory, the rate will be very high because no processing is required. For hardware buffers, the rate might be slower because the buffer might have to be downloaded to the sound card, which might have a limited transfer rate.

dwPlayCpuOverhead
Specifies the processing overhead as a percentage of main processing cycles needed to mix this sound buffer. For hardware buffers, this member will be 0

because the mixing is performed by the sound device. For software buffers, this member depends on the buffer format and the speed of the system processor.

Remarks

The **DSBCAPS** structure contains information similar to that found in the **DSBUFFERDESC** structure passed to the **IDirectSound8::CreateSoundBuffer** method, with some additional information. Additional information includes the location of the buffer (hardware or software) and some cost measures (such as the time to download the buffer if located in hardware, and the processing overhead to play the buffer if it is mixed in software).

The **dwFlags** member contains the same flags used by the **DSBUFFERDESC** structure. The only difference is that in the **DSBCAPS** structure, either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag is specified, according to the location of the buffer. In the **DSBUFFERDESC** structure, these flags are optional and are used to force the buffer to be located in either hardware or software.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound8::CreateSoundBuffer, **IDirectSoundBuffer8::GetCaps**

DSBPOSITIONNOTIFY

Describes a notification position. Used by the **IDirectSoundNotify8::SetNotificationPositions** method.

```
typedef struct {
    DWORD   dwOffset;
    HANDLE  hEventNotify;
} DSBPOSITIONNOTIFY, *LPDSBPOSITIONNOTIFY;

typedef const DSBPOSITIONNOTIFY *LPCDSBPOSITIONNOTIFY;
```

Members

dwOffset
Offset from the beginning of the buffer where the notify event is to be triggered, or DSBPN_OFFSETSTOP.

hEventNotify
Handle to the event to be signaled when the offset has been reached.

Remarks

The DSBPN_OFFSETSTOP value in the **dwOffset** member causes the event to be signaled when playback or capture stops, either because the end of the buffer has been reached (and playback or capture is not looping) or because the application called the **IDirectSoundBuffer8::Stop** or **IDirectSoundCaptureBuffer8::Stop** method.

When using DSBCAPS_LOCDEFER and DSBCAPS_NOTIFY along with any voice management flag, it is possible that a sound that has notifications set, but not yet reached, may be terminated by the voice manager. In this event, the notification event will not occur.

Requirements

Header: Declared in dsound.h.

DSBUFFERDESC

Describes the characteristics of a new DirectSoundBuffer object. Used by the **IDirectSound8::CreateSoundBuffer** method.

An earlier version of this structure, **DSBUFFERDESC1**, is maintained in Dsound.h for compatibility with DirectX 7 and earlier.

```
typedef struct {
    DWORD           dwSize;
    DWORD           dwFlags;
    DWORD           dwBufferBytes;
    DWORD           dwReserved;
    LPWAVEFORMATEX  lpwfxFormat;
    GUID            guid3DAlgorithm;
} DSBUFFERDESC, *LPDSBUFFERDESC;

typedef const DSBUFFERDESC *LPCDSBUFFERDESC;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Flags specifying the capabilities to include when creating a new DirectSoundBuffer object. The following values are defined. See also Remarks.

DSBCAPS_CTRL3D
The buffer has 3-D control capability. Cannot be combined with DSBCAPS_CTRLPAN, and cannot be set for a buffer with a stereo format in **lpwfxFormat**.

DSBCAPS_CTRLFREQUENCY

The buffer has frequency control capability. Cannot be combined with DSBCAPS_CTRLFX.

DSBCAPS_CTRLFX
The buffer supports effects processing. Cannot be combined with DSBCAPS_CTRLFREQUENCY. The wave format in **lpwfxFormat** must be an 8-bit or 16-bit PCM format with no more than two channels, and the buffer must be large enough to hold at least DSBSIZE_FX_MIN milliseconds of data.

DSBCAPS_CTRLPAN
The buffer has pan control capability. Cannot be combined with DSBCAPS_CTRL3D.

DSBCAPS_CTRLPOSITIONNOTIFY
The buffer has position notification capability. See Remarks.

DSBCAPS_CTRLVOLUME
The buffer has volume control capability.

DSBCAPS_GETCURRENTPOSITION2
The **IDirectSoundBuffer8::GetCurrentPosition** method uses the new behavior of the play cursor. In DirectSound in DirectX 1, the play cursor was significantly ahead of the actual playing sound on emulated sound cards; it was directly behind the write cursor. Now, if the DSBCAPS_GETCURRENTPOSITION2 flag is specified, the application can get a more accurate play cursor. If this flag is not specified, the old behavior is preserved for compatibility. Note that this flag affects only emulated sound cards. If a DirectSound driver is present, the play cursor is accurate for DirectSound in all versions of DirectX.

DSBCAPS_GLOBALFOCUS
The buffer is a global sound buffer. With this flag set, an application using DirectSound can continue to play its buffers if the user switches focus to another application, even if the new application uses DirectSound. The one exception is if you switch focus to a DirectSound application that uses the DSSCL_WRITEPRIMARY flag for its cooperative level. In this case, the global sounds from other applications will not be audible.

DSBCAPS_LOCDEFER
The buffer can be assigned to a hardware or software resource at play time. This flag must be set for buffers that use voice management.

DSBCAPS_LOCHARDWARE
The buffer must use hardware mixing, If the device does not support hardware mixing or if the required hardware memory is not available, the call to **IDirectSound8::CreateSoundBuffer** method will fail. The application must ensure that a mixing channel is available for this buffer; this condition is not guaranteed.

DSBCAPS_LOCSOFTWARE
The buffer must be in software memory and use software mixing, even if DSBCAPS_STATIC is specified and hardware resources are available.

DSBCAPS_MUTE3DATMAXDISTANCE

The sound is reduced to silence at the maximum distance. The buffer will stop playing when the maximum distance is exceeded, so that processor time is not wasted.

DSBCAPS_PRIMARYBUFFER
The buffer is a primary sound buffer. If this value is not specified, a secondary sound buffer is created. Cannot be combined with DSBCAPS_CTRLFX.

DSBCAPS_STATIC
The buffer is placed in on-board hardware memory, if available. No error occurs if such memory is not available. Compare this flag with DSBCAPS_LOCHARDWARE, which forces the buffer into hardware memory (probably system memory used by the driver for its mixing buffers) and causes an error if no such memory is available. This flag cannot be combined with DSBCAPS_CTRLFX.

DSBCAPS_STICKYFOCUS
The buffer has sticky focus. If the user switches to another application not using DirectSound, the application's normal buffers are muted, but sticky focus buffers are still audible.

dwBufferBytes
Size of the new buffer, in bytes. This value must be 0 when creating a buffer with the DSBCAPS_PRIMARYBUFFER flag. For secondary buffers, the minimum and maximum sizes allowed are specified by DSBSIZE_MIN and DSBSIZE_MAX, defined in Dsound.h.

dwReserved
Reserved. Must be 0.

lpwfxFormat
Address of a **WAVEFORMATEX** or **WAVEFORMATEXTENSIBLE** structure specifying the waveform format for the buffer. This value must be NULL for primary buffers. The application can use **IDirectSoundBuffer8::SetFormat** to set the format of the primary buffer.

guid3DAlgorithm
Unique identifier of the two-speaker virtualization algorithm to be used by the DirectSound3D HEL. If DSBCAPS_CTRL3D is not set in **dwFlags**, this member must be GUID_NULL (DS3DALG_DEFAULT). See also Remarks.

The following algorithm identifiers are defined:

DS3DALG_DEFAULT
DirectSound uses the default algorithm. In most cases this is DS3DALG_NO_VIRTUALIZATION. On WDM drivers, if the user has selected surround sound in Control Panel, the sound is panned among the left, center, right, and surround channels.

Applies to Software Mixing Only. Available on WDM or Vxd Drivers.

DS3DALG_NO_VIRTUALIZATION
3-D output is mapped onto normal left and right stereo panning. At 90 degrees to the left, the sound is coming out of only the left speaker; at 90 degrees to the right, sound is coming out of only the right speaker. The vertical axis is ignored except for scaling of volume due to distance. Doppler shift and

volume scaling are still applied, but the 3-D filtering is not performed on this buffer. This is the most efficient software implementation, but provides no virtual 3-D audio effect. When the DS3DALG_NO_VIRTUALIZATION flag is specified, HRTF processing will not be done. Note: Since DS3DALG_NO_VIRTUALIZATION uses only normal stereo panning, a buffer created with this algorithm may be accelerated by a 2-D hardware voice if no free 3-D hardware voices are available.

Applies to Software Mixing Only. Available on WDM or Vxd Drivers.

DS3DALG_HRTF_FULL

The 3-D API is processed with the high quality 3-D audio algorithm. This flag applies only to the software HEL. This algorithm gives the highest quality 3-D audio effect, but uses more CPU. See Remarks.

Applies to Software Mixing Only. Available on Microsoft® Windows® 98 Second Edition and Windows 2000 when using WDM drivers.

Note: If a sound buffer is created using one of the HRTF algorithms, and the HRTF algorithm is not available on the system (for example, a non-WDM system), a success code, DS_NO_VIRTUALIZATION, is returned. The sound buffer created will use DS3DALG_NO_VIRTUALIZATION instead. For this reason, applications should use the **SUCCEEDED** or **FAILED** macros rather than checking explicitly for DS_OK when calling **CreateSoundBuffer**.

DS3DALG_HRTF_LIGHT

The 3-D API is processed with the efficient 3-D audio algorithm. This flag applies only to the software HEL. This algorithm gives a good 3-D audio effect, but uses less CPU than DS3DALG_HRTF_FULL.

Applies to Software Mixing Only. Available on Windows 98 Second Edition and Windows 2000 when using WDM drivers.

Note: If a sound buffer is created using one of the HRTF algorithms, and the HRTF algorithm is not available on the system (for example, a non-WDM system), a success code, DS_NO_VIRTUALIZATION, is returned. The sound buffer created will use DS3DALG_NO_VIRTUALIZATION instead. For this reason, applications should use the **SUCCEEDED** or **FAILED** macros rather than checking explicitly for DS_OK when calling **CreateSoundBuffer**.

Remarks

When creating a primary buffer, applications must set the **dwBufferBytes** member to 0. DirectSound will determine the best buffer size for the particular sound device in use. To determine the size of a created primary buffer, call **IDirectSoundBuffer8::GetCaps**.

The DSBCAPS_CTRLDEFAULT flag is no longer supported. This flag was defined as DSBCAPS_CTRLPAN | DSBCAPS_CTRLVOLUME | DSBCAPS_CTRLFREQUENCY. By specifying only the flags you need, you cut down on unnecessary resource usage.

Sound buffers created with the DSBCAPS_CTRLPOSITIONNOTIFY flag must set a notification event. If you create a sound buffer with this flag but don't actually set any notifications, the behavior is undefined, and you might experience sounds being played back twice.

On VxD drivers, a sound buffer created with DSBCAPS_CTRLPOSITIONNOTIFY is always a software buffer, because the VxD driver model does not support notifications. With WDM drivers, a notification-enabled buffer can be in hardware, if hardware is available. Calling **IDirectSoundBuffer8::Play** with the DSBPLAY_LOCHARDWARE flag will fail when playing a buffer created with this flag set.

The DSBCAPS_LOCHARDWARE and DSBCAPS_LOCSOFTWARE flags are optional and mutually exclusive. DSBCAPS_LOCHARDWARE forces the buffer to reside in hardware, meaning that it will be mixed by the sound card. DSBCAPS_LOCSOFTWARE forces the buffer to reside in software, where it is mixed by the CPU. These flags are also defined for the **dwFlags** member of the **DSBCAPS** structure, where they indicate the actual location of the buffer.

The 3-D algorithms represent selection of the software emulation layer only—that is, the software algorithm that is used when no hardware is present for acceleration. In order to maximize hardware utilization, DS3DALG_NO_VIRTUALIZATION is treated as a special case. If no free 3-D hardware voices are available, the buffer is then treated as a 2-D buffer, but with 3-D API control. Specifically, when a sound buffer is created with DS3DALG_NO_VIRTUALIZATION, or when the buffer is played if the buffer was created with DSBPLAY_LOCDEFER, the following procedure is followed:

- If a free hardware 3-D voice is available, that 3-D hardware voice is used.
- If no free hardware 3-D voices are available and a 2-D hardware voice is available, that 2-D hardware voice will be used. This is possible because the DS3DALG_NO_VIRTUALIZATION algorithm is a simple stereo pan algorithm
- If no free 2-D or 3-D hardware voices are available, the voice will be played in software using the DS3DALG_NO_VIRTUALIZATION algorithm.

If a speaker configuration other than DSSPEAKER_HEADPHONE or DSSPEAKER_STEREO is in effect, the processing will be done as if for a two-speaker configuration.

If a buffer is created using one of the HRTF algorithms, and the HRTF algorithm is not available on the system (for example, a non-WDM system), a success code, DS_NO_VIRTUALIZATION, is returned. The sound buffer created will use DS3DALG_NO_VIRTUALIZATION instead. For this reason, applications should use the **SUCCEEDED** or **FAILED** macros rather than checking explicitly for DS_OK when calling **CreateSoundBuffer**.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound8::CreateSoundBuffer, DirectSound Buffers, Hardware Acceleration on ISA and PCI Cards

DSCAPS

Describes the capabilities of a DirectSound device. Used by the **IDirectSound8::GetCaps** method.

```
typedef {
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwMinSecondarySampleRate;
    DWORD  dwMaxSecondarySampleRate;
    DWORD  dwPrimaryBuffers;
    DWORD  dwMaxHwMixingAllBuffers;
    DWORD  dwMaxHwMixingStaticBuffers;
    DWORD  dwMaxHwMixingStreamingBuffers;
    DWORD  dwFreeHwMixingAllBuffers;
    DWORD  dwFreeHwMixingStaticBuffers;
    DWORD  dwFreeHwMixingStreamingBuffers;
    DWORD  dwMaxHw3DAllBuffers;
    DWORD  dwMaxHw3DStaticBuffers;
    DWORD  dwMaxHw3DStreamingBuffers;
    DWORD  dwFreeHw3DAllBuffers;
    DWORD  dwFreeHw3DStaticBuffers;
    DWORD  dwFreeHw3DStreamingBuffers;
    DWORD  dwTotalHwMemBytes;
    DWORD  dwFreeHwMemBytes;
    DWORD  dwMaxContigFreeHwMemBytes;
    DWORD  dwUnlockTransferRateHwBuffers;
    DWORD  dwPlayCpuOverheadSwBuffers;
    DWORD  dwReserved1;
    DWORD  dwReserved2;
} DSCAPS, *LPDSCAPS;

typedef const DSCAPS *LPCDSCAPS;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags

Flags describing device capabilities. Can include the following values.

DSCAPS_CERTIFIED

The driver has been tested and certified by Microsoft. This flag is always set for WDM drivers. To test for certification, use **IDirectSound8::VerifyCertification**.

DSCAPS_CONTINUOUSRATE
The device supports all sample rates between the **dwMinSecondarySampleRate** and **dwMaxSecondarySampleRate** member values. Typically, this means that the actual output rate will be within +/- 10 hertz (Hz) of the requested frequency.

DSCAPS_EMULDRIVER
The device does not have a DirectSound driver installed, so it is being emulated through the waveform-audio functions. Performance degradation should be expected.

DSCAPS_PRIMARY16BIT
The device supports primary sound buffers with 16-bit samples.

DSCAPS_PRIMARY8BIT
The device supports primary buffers with 8-bit samples.

DSCAPS_PRIMARYMONO
The device supports monophonic primary buffers.

DSCAPS_PRIMARYSTEREO
The device supports stereo primary buffers.

DSCAPS_SECONDARY16BIT
The device supports hardware-mixed secondary sound buffers with 16-bit samples.

DSCAPS_SECONDARY8BIT
The device supports hardware-mixed secondary buffers with 8-bit samples.

DSCAPS_SECONDARYMONO
The device supports hardware-mixed monophonic secondary buffers.

DSCAPS_SECONDARYSTEREO
The device supports hardware-mixed stereo secondary buffers.

dwMinSecondarySampleRate, dwMaxSecondarySampleRate
Minimum and maximum sample rate specifications that are supported by this device's hardware secondary sound buffers.

dwPrimaryBuffers
Number of primary buffers supported. This value will always be 1.

dwMaxHwMixingAllBuffers
Number of buffers that can be mixed in hardware. This member can be less than the sum of **dwMaxHwMixingStaticBuffers** and **dwMaxHwMixingStreamingBuffers**. Resource tradeoffs frequently occur.

dwMaxHwMixingStaticBuffers
Maximum number of static buffers.

dwMaxHwMixingStreamingBuffers
Maximum number of streaming sound buffers.

dwFreeHwMixingAllBuffers

Number of unallocated buffers. On WDM drivers, this includes **dwFreeHw3DAllBuffers**.

dwFreeHwMixingStaticBuffers
Number of unallocated static buffers.

dwFreeHwMixingStreamingBuffers
Number of unallocated streaming buffers.

dwMaxHw3DAllBuffers
Maximum number of 3-D buffers.

dwMaxHw3DStaticBuffers
Maximum number of static 3-D buffers.

dwMaxHw3DStreamingBuffers
Maximum number of streaming 3-D buffers.

dwFreeHw3DAllBuffers
Number of unallocated 3-D buffers.

dwFreeHw3DStaticBuffers
Number of unallocated static 3-D buffers.

dwFreeHw3DStreamingBuffers
Number of unallocated streaming 3-D buffers.

dwTotalHwMemBytes
Size, in bytes, of the amount of memory on the sound card that stores static sound buffers.

dwFreeHwMemBytes
Size, in bytes, of the free memory on the sound card.

dwMaxContigFreeHwMemBytes
Size, in bytes, of the largest contiguous block of free memory on the sound card.

dwUnlockTransferRateHwBuffers
Description of the rate, in kilobytes per second, at which data can be transferred to hardware static sound buffers. This and the number of bytes transferred determines the duration of a call to the **IDirectSoundBuffer8::Unlock** method.

dwPlayCpuOverheadSwBuffers
Description of the processing overhead, as a percentage of the central processing unit, needed to mix software buffers (those located in main system memory). This varies according to the bus type, the processor type, and the clock speed.

The unlock transfer rate for software buffers is 0 because the data need not be transferred anywhere. Similarly, the play processing overhead for hardware buffers is 0 because the mixing is done by the sound device.

dwReserved1
Not used.

dwReserved2
Not used.

Remarks

Some audio cards may be unable to report accurately the number of available or free hardware buffers. This can happen, for example, when the card can play more sounds at lower sampling rates than at higher rates. In general, a nonzero value in any of the members relating to number of free hardware buffers signifies that at least one hardware resource of the appropriate type is available.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSound8::GetCaps

DSCBCAPS

Describes the capabilities of a capture buffer. Used by the **IDirectSoundCaptureBuffer8::GetCaps** method.

```
typedef struct {
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwBufferBytes;
    DWORD  dwReserved;
} DSCBCAPS, *LPDSCBCAPS;

typedef const DSCBCAPS *LPCDSCBCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Specifies device capabilities. Can be zero or the following flag:

DSCBCAPS_WAVEMAPPED
The Win32 wave mapper will be used for formats not supported by the device.

dwBufferBytes
The size, in bytes, of the capture buffer.

dwReserved
Reserved for future use.

Requirements

Header: Declared in dsound.h.

DSCBUFFERDESC

Describes a capture buffer. Used by the **IDirectSoundCapture8::CreateCaptureBuffer** method.

An earlier version of this structure, **DSCBUFFERDESC1**, is maintained in Dsound.h for compatibility with DirectX 7 and earlier.

```
typedef struct {
    DWORD           dwSize;
    DWORD           dwFlags;
    DWORD           dwBufferBytes;
    DWORD           dwReserved;
    LPWAVEFORMATEX  lpwfxFormat;
    DWORD           dwFXCount;
    LPDSCEFFECTDESC lpDSCFXDesc;
} DSCBUFFERDESC, *LPDSCBUFFERDESC;

typedef const DSCBUFFERDESC *LPCDSCBUFFERDESC;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Specifies device capabilities. Can be zero or one or more of the following flag:

DSCBCAPS_CTRLFX
The buffer supports effects. See Remarks.

DSCBCAPS_WAVEMAPPED
The Win32 wave mapper will be used for formats not supported by the device.

dwBufferBytes
Size of capture buffer to create, in bytes.

dwReserved
Reserved for future use.

lpwfxFormat
Pointer to a **WAVEFORMATEX** structure containing the format in which to capture the data.

dwFXCount
Number of elements in the array pointed to by **lpDSCFXDesc**. Must be zero unless DSCBCAPS_CTRLFX is present in **dwFlags**.

lpDSCFXDesc
Address of an array of **DSCEFFECTDESC** structures that describe effects supported by hardware for the buffer. User-mode DMOs are not supported.

Remarks

The DSCBCAPS_CTRLFX flag is supported only on buffers created by an object of class CLSID_DirectSoundCapture8. If the **IDirectSoundCapture8** interface was obtained from **DirectSoundCaptureCreate8**, this flag is supported; if it was obtained from the earlier **DirectSoundCaptureCreate** function, it is not.

Capture effects are not supported on current operating systems.

Requirements

Header: Declared in dsound.h.

DSCCAPS

Describes the capabilities of the capture device. Used by the **IDirectSoundCapture8::GetCaps** method.

```
typedef struct {
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwFormats;
    DWORD  dwChannels;
} DSCCAPS, *LPDSCCAPS;

typedef const DSCCAPS *LPCDSCCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Specifies device capabilities. Can be 0 or the following flags:

DSCCAPS_EMULDRIVER
There is no DirectSoundCapture driver for the device, so the standard wave audio functions are being used.

DSCCAPS_CERTIFIED
The driver for the device is a certified WDM driver.

DSCCAPS_MULTIPLECAPTURE
The capture device can be opened by multiple applications which can all receive valid capture data simultaneously.

dwFormats
Standard formats that are supported. These are equivalent to the values in the **WAVEINCAPS** structure used in the Win32 waveform audio functions, and are reproduced here for convenience.

Value	Meaning

WAVE_FORMAT_1M08	11.025 kHz, mono, 8-bit
WAVE_FORMAT_1M16	11.025 kHz, mono, 16-bit
WAVE_FORMAT_1S08	11.025 kHz, stereo, 8-bit
WAVE_FORMAT_1S16	11.025 kHz, stereo, 16-bit
WAVE_FORMAT_2M08	22.05 kHz, mono, 8-bit
WAVE_FORMAT_2M16	22.05 kHz, mono, 16-bit
WAVE_FORMAT_2S08	22.05 kHz, stereo, 8-bit
WAVE_FORMAT_2S16	22.05 kHz, stereo, 16-bit
WAVE_FORMAT_4M08	44.1 kHz, mono, 8-bit
WAVE_FORMAT_4M16	44.1 kHz, mono, 16-bit
WAVE_FORMAT_4S08	44.1 kHz, stereo, 8-bit
WAVE_FORMAT_4S16	44.1 kHz, stereo, 16-bit

dwChannels

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

Requirements

Header: Declared in dsound.h.

DSCEFFECTDESC

Contains parameters for an effect associated with a capture buffer.

```
typedef struct _DSCEFFECTDESC {
    DWORD       dwSize;
    DWORD       dwFlags;
    GUID        guidDSCFXClass;
    GUID        guidDSCFXInstance;
    DWORD       dwReserved1;
    DWORD       dwReserved2;
} DSCEFFECTDESC, *LPDSCEFFECTDESC;

typedef const DSCEFFECTDESC *LPCDSCEFFECTDESC;
```

Members

dwSize

Size of the structure, in bytes.

dwFlags

Flags that specify desired parameters of the effect. When this structure is passed to **IDirectSoundCapture8::CreateCaptureBuffer**, this member can be zero or one or more of the following values.

DSCFX_LOCHARDWARE
Effect specified by **guidDSCFXInstance** must be in hardware.

DSCFX_LOCSOFTWARE
Effect specified by **guidDSCFXInstance** must be in software.

On return, this member can contain one or more of the following values.

DSCFXR_LOCHARDWARE
Effect was created in hardware.

DSCFXR_LOCSOFTWARE
Effect was created in software.

DSCFXR_UNALLOCATED
The effect was not created.

DSCFXR_FAILED
Effect creation failed.

DSCFXR_UNKNOWN
Result of effect creation is unknown.

guidDSCFXClass
Value of type **GUID** that specifies the class identifier of the effect. The following standard identifiers are defined.

DSCFX_CLASS_AEC
Acoustic echo cancellation.

DSCFX_CLASS_NS
Noise suppression.

guidDSCFXInstance
Value of type **GUID** that specifies the unique identifier of the preferred effect. The following standard identifiers are defined.

DSCFX_MS_AEC
Microsoft acoustic echo cancellation. Available in software only.

DSCFX_MS _NS
Microsoft noise suppression. Available in software only.

DSCFX_SYSTEM_AEC
System default acoustic echo cancellation.

DSCFX_SYSTEM_NS
System default noise suppression.

DSCFX_NOP
No effect.

dwReserved1
Reserved. Must be 0.

dwReserved2
Reserved. Must be 0.

Requirements

Header: Declared in dsound.h.

DSCFXAec

Contains parameters for acoustic echo cancellation in a capture buffer.

```
typedef struct _DSCFXAec {
    BOOL  fEnable;
    BOOL  fReset;
} DSCFXAec, *LPDSCFXAec;

typedef const DSCFXAec *LPCDSCFXAec;
```

Members

fEnable
Boolean value that specifies whether the effect is enabled.

fReset
Boolean value that specifies whether to reset the effect to the default state.

Remarks

Applications should not reset an effect except when necessary because it has entered an incorrect state. This might be done in response to user input. The **fReset** flag must not be set arbitrarily at startup, because another application might be using the same effect.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundCaptureFXAec8::GetAllParameters,
IDirectSoundCaptureFXAec8::SetAllParameters

DSCFXNoiseSuppress

Contains parameters for noise suppression in a capture buffer.

```
typedef struct _DSCFXNoiseSuppress {
    BOOL  fEnable;
    BOOL  fReset;
} DSCFXNoiseSuppress, *LPDSCFXNoiseSuppress;
```

```
typedef const DSCFXNoiseSuppress *LPCDSCFXNoiseSuppress;
```

Members

fEnable
Boolean value that specifies whether the effect is enabled.

fReset
Boolean value that specifies whether to reset the effect to the default state.

Remarks

Applications should not reset an effect except when necessary because it has entered an incorrect state. This might be done in response to user input. The **fReset** flag must not be set arbitrarily at startup, because another application might be using the same effect.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundCaptureFXNoiseSuppress8::GetAllParameters,
IDirectSoundCaptureFXNoiseSuppress8::SetAllParameters

DSEFFECTDESC

Describes an effect associated with a buffer.

```
typedef struct _DSEFFECTDESC {
    DWORD  dwSize;
    DWORD  dwFlags;
    GUID   guidDSFXClass;
    DWORD  dwReserved1;
    DWORD  dwReserved2
} DSEFFECTDESC, *LPDSEFFECTDESC;

typedef const DSEFFECTDESC *LPCDSEFFECTDESC;
```

Members

dwSize
Size of the structure, in bytes.

dwFlags
Flags. Can be 0 or one of the following values:

DSFX_LOCHARDWARE
Effect must be in hardware. If the effect is not available in hardware, **IDirectSoundBuffer8::SetFX** raises an error. Because DirectX 8.0 does not support hardware acceleration of effects, this flag should not be used.

DSFX_LOCSOFTWARE
Effect must be in software, even if the hardware supports acceleration of **guidDSFXClass**. If the effect is not available in software, **SetFX** raises an error. In DirectX 8.0, all effects are in software regardless of whether this flag is set.

guidDSFXClass
Class identifier of the effect. The following standard effect classes are defined.

GUID_DSFX_STANDARD_CHORUS
GUID_DSFX_STANDARD_COMPRESSOR
GUID_DSFX_STANDARD_DISTORTION
GUID_DSFX_STANDARD_ECHO
GUID_DSFX_STANDARD_FLANGER
GUID_DSFX_STANDARD_GARGLE
GUID_DSFX_STANDARD_I3DL2REVERB
GUID_DSFX_STANDARD_PARAMEQ
GUID_DSFX_STANDARD_WAVES_REVERB

dwReserved1
Reserved for future use.

dwReserved2
Reserved for future use.

Remarks

If **dwFlags** is zero, the effect is placed in hardware if possible. If the hardware does not support the effect, software is used. If the effect is not available at all, the call to **SetFX** fails.

An effect of class GUID_DSFX_STANDARD_WAVES_REVERB can be set only on a buffer that has a 16-bit audio format.

Requirements

Header: Declared in dsound.h.

See Also

DSBCAPS, **IDirectSoundBuffer8::SetFX**, Using Effects

DSFXI3DL2Reverb

Contains parameters for an I3DL2 (Interactive 3D Audio Level 2) reverberation effect.

```
typedef struct _DSFXI3DL2Reverb {
    LONG    lRoom;
    LONG    lRoomHF;
    FLOAT   flRoomRolloffFactor;
    FLOAT   flDecayTime;
    FLOAT   flDecayHFRatio;
    LONG    lReflections;
    FLOAT   flReflectionsDelay;
    LONG    lReverb;
    FLOAT   flReverbDelay;
    FLOAT   flDiffusion;
    FLOAT   flDensity;
    FLOAT   flHFReference;
} DSFXI3DL2Reverb, *LPDSFXI3DL2Reverb;

typedef const DSFXI3DL2Reverb *LPCDSFXI3DL2Reverb;
```

Members

lRoom
Attenuation of the room effect, in millibels (mB), in the range from DSFX_I3DL2REVERB_ROOM_MIN to DSFX_I3DL2REVERB_ROOM_MAX. The default value is DSFX_I3DL2REVERB_ROOM_DEFAULT, or -1000 mB.

lRoomHF
Attenuation of the room high-frequency effect, in mB, in the range from DSFX_I3DL2REVERB_ROOMHF_MIN to DSFX_I3DL2REVERB_ROOMHF_MAX. The default value is DSFX_I3DL2REVERB_ROOMHF_DEFAULT, or 0 mB.

flRoomRolloffFactor
Rolloff factor for the reflected signals, in the range from DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_MIN to DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_MAX. The default value is DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_DEFAULT, or 0.0. The rolloff factor for the direct path is controlled by the DirectSound listener.

flDecayTime
Decay time, in seconds, in the range from DSFX_I3DL2REVERB_DECAYTIME_MIN to DSFX_I3DL2REVERB_DECAYTIME_MAX. The default value is DSFX_I3DL2REVERB_DECAYTIME_DEFAULT, or 1.49 second.

flDecayHFRatio
Ratio of the decay time at high frequencies to the decay time at low frequencies, in the range from DSFX_I3DL2REVERB_DECAYHFRATIO_MIN to DSFX_I3DL2REVERB_DECAYHFRATIO_MAX. The default value is DSFX_I3DL2REVERB_DECAYHFRATIO_DEFAULT, or 0.83.

lReflections
Attenuation of early reflections relative to **lRoom**, in mB, in the range from DSFX_I3DL2REVERB_REFLECTIONS_MIN to DSFX_I3DL2REVERB_REFLECTIONS_MAX. The default value is DSFX_I3DL2REVERB_REFLECTIONS_DEFAULT, or -2602 mB.

flReflectionsDelay
Delay time of the first reflection relative to the direct path, in seconds, in the range from DSFX_I3DL2REVERB_REFLECTIONSDELAY_MIN to DSFX_I3DL2REVERB_REFLECTIONSDELAY_DEFAULT. The default value is 0.007 seconds.

lReverb
Attenuation of late reverberation relative to **lRoom**, in mB, in the range -from DSFX_I3DL2REVERB_REVERB_MIN to DSFX_I3DL2REVERB_REVERB_MAX. The default value is DSFX_I3DL2REVERB_REVERB_DEFAULT, or 200 mB.

flReverbDelay
Time limit between the early reflections and the late reverberation relative to the time of the first reflection, in seconds, in the range from DSFX_I3DL2REVERB_REVERBDELAY_MIN to DSFX_I3DL2REVERB_REVERBDELAY_MAX. The default value is DSFX_I3DL2REVERB_REVERBDELAY_DEFAULT, or 0.011 seconds.

flDiffusion
Echo density in the late reverberation decay, in percent, in the range from DSFX_I3DL2REVERB_DIFFUSION_MIN to DSFX_I3DL2REVERB_DIFFUSION_MAX. The default value is DSFX_I3DL2REVERB_DIFFUSION_DEFAULT, or 100.0 percent.

flDensity
Modal density in the late reverberation decay, in percent, in the range from DSFX_I3DL2REVERB_DENSITY_MIN to DSFX_I3DL2REVERB_DENSITY_MAX. The default value is DSFX_I3DL2REVERB_DENSITY_DEFAULT, or 100.0 percent.

flHFReference
Reference high frequency, in hertz, in the range from DSFX_I3DL2REVERB_HFREFERENCE_MIN to DSFX_I3DL2REVERB_HFREFERENCE_MAX. The default value is DSFX_I3DL2REVERB_HFREFERENCE_DEFAULT, or 5000.0 Hz.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXI3DL2Reverb8

DSFXChorus

Contains parameters for a chorus effect.

```
typedef struct _DSFXChorus {
    FLOAT       fWetDryMix;
    FLOAT       fDepth;
    FLOAT       fFeedback;
    FLOAT       fFrequency;
    LONG        lWaveform;
    FLOAT       fDelay;
    LONG        lPhase;
} DSFXChorus, *LPDSFXChorus;

typedef const DSFXChorus *LPCDSFXChorus;
```

Members

fWetDryMix
Ratio of wet (processed) signal to dry (unprocessed) signal. Must be in the range from DSFXCHORUS_WETDRYMIX_MIN through DSFXCHORUS_WETDRYMIX_MAX (all wet).

fDepth
Percentage by which the delay time is modulated by the low-frequency oscillator, in hundredths of a percentage point. Must be in the range from DSFXCHORUS_DEPTH_MIN through DSFXCHORUS_DEPTH_MAX. The default value is 25.

fFeedback
Percentage of output signal to feed back into the effect's input, in the range from DSFXCHORUS_FEEDBACK_MIN to DSFXCHORUS_FEEDBACK_MAX. The default value is 0.

fFrequency
Frequency of the LFO, in the range from DSFXCHORUS_FREQUENCY_MIN to DSFXCHORUS_FREQUENCY_MAX. The default value is 0.

lWaveform
Waveform of the LFO. Defined values are DSFXCHORUS_WAVE_TRIANGLE and DSFXCHORUS_WAVE_SIN. By default, the waveform is a sine.

fDelay

Number of milliseconds the input is delayed before it is played back, in the range from DSFXCHORUS_DELAY_MIN to DSFXCHORUS_DELAY_MAX. The default value is 0 ms.

lPhase

Phase differential between left and right LFOs, in the range from DSFXCHORUS_PHASE_MIN through DSFXCHORUS_PHASE_MAX. Possible values are defined as follows:

DSFXCHORUS_PHASE_NEG_180
DSFXCHORUS_PHASE_NEG_90
DSFXCHORUS_PHASE_ZERO
DSFXCHORUS_PHASE_90
DSFXCHORUS_PHASE_180

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXChorus8, Chorus

DSFXCompressor

Contains parameters for a compression effect.

```
typedef struct _DSFXCompressor {
    FLOAT   fGain;
    FLOAT   fAttack;
    FLOAT   fRelease;
    FLOAT   fThreshold;
    FLOAT   fRatio;
    FLOAT   fPredelay;
} DSFXCompressor, *LPDSFXCompressor;

typedef const DSFXCompressor *LPCDSFXCompressor;
```

Members

fGain

Output gain of signal after compression, in the range from DSFXCOMPRESSOR_GAIN_MIN to DSFXCOMPRESSOR_GAIN_MAX. The default value is 0 dB.

fAttack

Time before compression reaches its full value, in the range from DSFXCOMPRESSOR_ATTACK_MIN to DSFXCOMPRESSOR_ATTACK_MAX. The default value is 0.01 ms.

fRelease

Speed at which compression is stopped after input drops below **fThreshold**, in the range from DSFXCOMPRESSOR_RELEASE_MIN to DSFXCOMPRESSOR_RELEASE_MAX. The default value is 50 ms.

fThreshold

Point at which compression begins, in decibels, in the range from DSFXCOMPRESSOR_THRESHOLD_MIN to DSFXCOMPRESSOR_THRESHOLD_MAX. The default value is -10 dB.

fRatio

Compression ratio, in the range from DSFXCOMPRESSOR_RATIO_MIN to DSFXCOMPRESSOR_RATIO_MAX. The default value is 10, which means 10:1 compression.

fPredelay

Time after **lThreshold** is reached before attack phase is started, in milliseconds, in the range from DSFXCOMPRESSOR_PREDELAY_MIN to DSFXCOMPRESSOR_PREDELAY_MAX. The default value is 0 ms.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXCompressor8, Compression

DSFXDistortion

Contains parameters for a distortion effect.

```
typedef struct _DSFXDistortion {
    FLOAT   fGain;
    FLOAT   fEdge;
    FLOAT   fPostEQCenterFrequency;
    FLOAT   fPostEQBandwidth;
    FLOAT   fPreLowpassCutoff;
} DSFXDistortion, *LPDSFXDistortion;

typedef const DSFXDistortion *LPCDSFXDistortion;
```

Members

fGain

Amount of signal change after distortion, in the range from DSFXDISTORTION_GAIN_MIN through DSFXDISTORTION_GAIN_MAX. The default value is 0 dB.

fEdge

Percentage of distortion intensity, in the range in the range from DSFXDISTORTION_EDGE_MIN through DSFXDISTORTION_EDGE_MAX. The default value is 50 percent.

fPostEQCenterFrequency

Center frequency of harmonic content addition, in the range from DSFXDISTORTION_POSTEQCENTERFREQUENCY_MIN through DSFXDISTORTION_POSTEQCENTERFREQUENCY_MAX. The default value is 4000 Hz.

fPostEQBandwidth

Width of frequency band that determines range of harmonic content addition, in the range from DSFXDISTORTION_POSTEQBANDWIDTH_MIN through DSFXDISTORTION_POSTEQBANDWIDTH_MAX. The default value is 4000 Hz.

fPreLowpassCutoff

Filter cutoff for high-frequency harmonics attenuation, in the range from DSFXDISTORTION_PRELOWPASSCUTOFF_MIN through DSFXDISTORTION_PRELOWPASSCUTOFF_MAX. The default value is 4000 Hz.

Remarks

The values in **fPostEQBandwidth**, **fPostEQCenterFrequency**, and **fPreLowpassCutoff** cannot exceed one-third of the frequency of the buffer.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXDistortion8, Distortion

DSFXEcho

Contains parameters for an echo effect.

```
typedef struct _DSFXEcho {
    FLOAT   fWetDryMix;
    FLOAT   fFeedback;
    FLOAT   fLeftDelay;
    FLOAT   fRightDelay;
    LONG    lPanDelay;
} DSFXEcho, *LPDSFXEcho;

typedef const DSFXEcho *LPCDSFXEcho;
```

Members

fWetDryMix
Ratio of wet (processed) signal to dry (unprocessed) signal. Must be in the range from DSFXECHO_WETDRYMIX_MIN through DSFXECHO_WETDRYMIX_MAX (all wet).

fFeedback
Percentage of output fed back into input, in the range from DSFXECHO_FEEDBACK_MIN through DSFXECHO_FEEDBACK_MAX. The default value is 0.

fLeftDelay
Delay for left channel, in milliseconds, in the range from DSFXECHO_LEFTDELAY_MIN through DSFXECHO_LEFTDELAY_MAX. The default value is 333 ms.

fRightDelay
Delay for right channel, in milliseconds, in the range from DSFXECHO_RIGHTDELAY_MIN through DSFXECHO_RIGHTDELAY_MAX. The default value is 333 ms.

lPanDelay
Value that specifies whether to swap left and right delays with each successive echo. The default value is zero, meaning no swap. Possible values are defined as DSFXECHO_PANDELAY_MIN (equivalent to FALSE) and DSFXECHO_PANDELAY_MAX (equivalent to TRUE).

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXEcho8, Echo

DSFXFlanger

Contains parameters for a flange effect.

```
typedef struct _DSFXFlanger {
    FLOAT   fWetDryMix;
    FLOAT   fDepth;
    FLOAT   fFeedback;
    FLOAT   fFrequency;
    LONG    lWaveform;
    FLOAT   fDelay;
    LONG    lPhase;
} DSFXFlanger, *LPDSFXFlanger;
```

```
typedef const DSFXFlanger *LPCDSFXFlanger;
```

Members

fWetDryMix
Ratio of wet (processed) signal to dry (unprocessed) signal. Must be in the range from DSFXFLANGER_WETDRYMIX_MIN through DSFXFLANGER_WETDRYMIX_MAX (all wet).

fDepth
Percentage by which the delay time is modulated by the low-frequency oscillator (LFO), in hundredths of a percentage point. Must be in the range from DSFXFLANGER_DEPTH_MIN through DSFXFLANGER_DEPTH_MAX. The default value is 25.

fFeedback
Percentage of output signal to feed back into the effect's input, in the range from DSFXFLANGER_FEEDBACK_MIN to DSFXFLANGER_FEEDBACK_MAX. The default value is 0.

fFrequency
Frequency of the LFO, in the range from DSFXFLANGER_FREQUENCY_MIN to DSFXFLANGER_FREQUENCY_MAX. The default value is 0.

lWaveform
Waveform of the LFO. By default, the waveform is a sine. Possible values are defined as follows:

DSFXFLANGER_WAVE_TRIANGLE
Triangle.

DSFXFLANGER_WAVE_SIN
Sine.

fDelay
Number of milliseconds the input is delayed before it is played back, in the range from DSFXFLANGER_DELAY_MIN to DSFXFLANGER_DELAY_MAX. The default value is 0 ms.

lPhase
Phase differential between left and right LFOs, in the range from DSFXFLANGER_PHASE_MIN through DSFXFLANGER_PHASE_MAX. Possible values are defined as follows:

DSFXFLANGER_PHASE_NEG_180
DSFXFLANGER_PHASE_NEG_90
DSFXFLANGER_PHASE_ZERO
DSFXFLANGER_PHASE_90
DSFXFLANGER_PHASE_180

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXFlanger8, Flange

DSFXGargle

Contains parameters for an amplitude modulation effect.

```
typedef struct _DSFXGargle {
    DWORD  dwRateHz;
    DWORD  dwWaveShape;
} DSFXGargle, *LPDSFXGargle;

typedef const DSFXGargle *LPCDSFXGargle;
```

Members

dwRateHz
Rate of modulation, in Hertz. Must be in the range from DSFXGARGLE_RATEHZ_MIN through DSFXGARGLE_RATEHZ_MAX.

dwWaveShape
Shape of the modulation wave. The following values are defined.

DSFXGARGLE_WAVE_TRIANGLE
Triangular wave.

DSFXGARGLE_WAVE_SQUARE
Square wave.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXGargle8, Gargle

DSFXParamEq

Contains parameters for a parametric equalizer effect.

```
typedef struct _DSFXParamEq {
    FLOAT fCenter;
    FLOAT fBandwidth;
    FLOAT fGain;
} DSFXParamEq, *LPDSFXParamEq;
```

Members

fCenter
Center frequency, in hertz, in the range from DSFXPARAMEQ_CENTER_MIN to DSFXPARAMEQ_CENTER_MAX. This value cannot exceed one-third of the frequency of the buffer.

fBandwidth
Bandwidth, in semitones, in the range from DSFXPARAMEQ_BANDWIDTH_MIN to DSFXPARAMEQ_BANDWIDTH_MAX.

fGain
Gain, in the range from DSFXPARAMEQ_GAIN_MIN to DSFXPARAMEQ_GAIN_MAX.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXParamEq8, Parametric Equalizer

DSFXWavesReverb

Contains parameters for a wave reverberation effect.

```
typedef struct _DSFXWavesReverb {
    FLOAT   fInGain;
    FLOAT   fReverbMix;
    FLOAT   fReverbTime;
    FLOAT   fHighFreqRTRatio;
} DSFXWavesReverb, *LPDSFXWavesReverb;

typedef const DSFXWavesReverb *LPCDSFXWavesReverb;
```

Members

fInGain
Input gain of signal, in decibels (dB), in the range from DSFX_WAVESREVERB_INGAIN_MIN through DSFX_WAVESREVERB_INGAIN_MAX. The default value is DSFX_WAVESREVERB_INGAIN_DEFAULT, or 0 dB.

fReverbMix
Reverb mix, in dB, in the range from DSFX_WAVESREVERB_REVERBMIX_MIN through

DSFX_WAVESREVERB_REVERBMIX_MAX. The default value is DSFX_WAVESREVERB_REVERBMIX_DEFAULT, or 0 dB.

fReverbTime

Reverb time, in milliseconds, in the range from DSFX_WAVESREVERB_REVERBTIME_MIN through DSFX_WAVESREVERB_REVERBTIME_MAX. The default value is DSFX_WAVESREVERB_REVERBTIME_DEFAULT, or 1000.

fHighFreqRTRatio

In the range from DSFX_WAVESREVERB_HIGHFREQRTRATIO_MIN through DSFX_WAVESREVERB_HIGHFREQRTRATIO_MAX. The default value is DSFX_WAVESREVERB_HIGHFREQRTRATIO_DEFAULT, or 0.001.

Requirements

Header: Declared in dsound.h.

See Also

IDirectSoundFXWavesReverb8

WAVEFORMATEX

Defines the format of waveform-audio data. Only format information common to all waveform-audio data formats is included in this structure. For formats that require additional information, this structure is included as the first member in another structure, along with the additional information.

This structure is part of the Platform SDK and is not declared in Dsound.h. It is documented here for convenience.

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

Members

wFormatTag

Waveform-audio format type. Format tags are registered with Microsoft Corporation for many compression algorithms. A complete list of format tags can

be found in the Mmreg.h header file. For one- or two-channel PCM data, this value should be WAVEFORMAT_PCM.

nChannels
Number of channels in the waveform-audio data. Monaural data uses one channel and stereo data uses two channels.

nSamplesPerSec
Sample rate, in samples per second (hertz). If **wFormatTag** is WAVE_FORMAT_PCM, then common values for nSamplesPerSec are 8.0 kHz, 11.025 kHz, 22.05 kHz, and 44.1 kHz. For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

nAvgBytesPerSec
Required average data-transfer rate, in bytes per second, for the format tag. If **wFormatTag** is WAVE_FORMAT_PCM, **nAvgBytesPerSec** should be equal to the product of **nSamplesPerSec** and **nBlockAlign**. For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

nBlockAlign
Block alignment, in bytes. The block alignment is the minimum atomic unit of data for the **wFormatTag** format type. If **wFormatTag** is WAVE_FORMAT_PCM or WAVE_FORMAT_EXTENSIBLE, **nBlockAlign** must be equal to the product of **nChannels** and **wBitsPerSample** divided by 8 (bits per byte). For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

Software must process a multiple of **nBlockAlign** bytes of data at a time. Data written and read from a device must always start at the beginning of a block. For example, it is illegal to start playback of PCM data in the middle of a sample (that is, on a non-block-aligned boundary).

wBitsPerSample
Bits per sample for the **wFormatTag** format type. If **wFormatTag** is WAVE_FORMAT_PCM, then **wBitsPerSample** should be equal to 8 or 16. If **wFormatTag** is WAVE_FORMAT_EXTENSIBLE, this value can be any integer multiple of 8.

cbSize
Size, in bytes, of extra format information appended to the end of the **WAVEFORMATEX** structure. This information can be used by non-PCM formats to store extra attributes for the **wFormatTag**. If no extra information is required by the **wFormatTag**, this member must be set to zero. For WAVE_FORMAT_PCM formats, this member is ignored.

Requirements

Header: Declared in Mmreg.h.

See Also

WAVEFORMATEXTENSIBLE

WAVEFORMATEXTENSIBLE

Defines the format of waveform-audio data for formats having more than two channels. This structure is part of the Platform SDK and is not declared in Dsound.h. It is documented here for convenience.

```
typedef struct {
    WAVEFORMATEX  Format;
    union {
        WORD     wValidBitsPerSample;
        WORD     wSamplesPerBlock;
        WORD     wReserved;
    } Samples;
    DWORD        dwChannelMask;
    GUID         SubFormat;
} WAVEFORMATEXTENSIBLE, *PWAVEFORMATEXTENSIBLE;
```

Members

Format
WAVEFORMATEX structure that specifies the basic format. The **wFormatTag** member must be WAVE_FORMAT_EXTENSIBLE, defined in Mmreg.h. The **cbSize** member must be at least 22.

wValidBitsPerSample
How many bits of precision are present in the signal. Usually equal to **WAVEFORMATEX.wBitsPerSample**.

wSamplesPerBlock
How many samples are contained in one compressed block of audio data. If zero, the number of samples in each block is variable.

wReserved
Set to zero if neither **wValidBitsPerSample** nor **wSamplesPerBlock** applies to the format.

dwChannelMask
Bitmask specifying which channels are present. See Remarks.

SubFormat
Subformat of the data. Must be KSDATAFORMAT_SUBTYPE_PCM, defined in Ksmedia.h.

Remarks

The **dwChannelMask** member indicates which channels are present in the multichannel stream. The least significant bit corresponds with the front left speaker, the next least significant bit corresponds to the front right speaker, and so on. The bits, in order of significance, are defined in Ksmedia.h and Mmreg.h as follows:

SPEAKER_FRONT_LEFT
SPEAKER_FRONT_RIGHT
SPEAKER_FRONT_CENTER
SPEAKER_LOW_FREQUENCY
SPEAKER_BACK_LEFT
SPEAKER_BACK_RIGHT
SPEAKER_FRONT_LEFT_OF_CENTER
SPEAKER_FRONT_RIGHT_OF_CENTER
SPEAKER_BACK_CENTER
SPEAKER_SIDE_LEFT
SPEAKER_SIDE_RIGHT
SPEAKER_TOP_CENTER
SPEAKER_TOP_FRONT_LEFT
SPEAKER_TOP_FRONT_CENTER
SPEAKER_TOP_FRONT_RIGHT
SPEAKER_TOP_BACK_LEFT
SPEAKER_TOP_BACK_CENTER
SPEAKER_TOP_BACK_RIGHT

For more information on this structure, see the document "Enhanced Audio Formats for Multi-Channel Configurations and High-Bit Resolution", available at http://www.microsoft.com/hwdev/audio/multichaud.htm.

Requirements

Header: Declared in Mmreg.h, Ksproxy.h.

DirectSound Return Values

This section provides a brief explanation of the various error codes that can be returned by DirectSound methods. For a list of the specific codes that each method can return, see the individual method descriptions. The lists given there are not necessarily comprehensive.

Error codes are presented in the following sections:

- DirectSound Return Values By Number
- DirectSound Return Values By Name

DirectSound Return Values By Number

The following table lists DirectSound return values sorted by hexadecimal value. For a description, click on the constant.

Hexadecimal	Constant
0x00000000	DS_OK
0x00000007	DSERR_OUTOFMEMORY
0x000001AE	DSERR_NOINTERFACE
0x0878000A	DS_NO_VIRTUALIZATION
0x08780014	DS_INCOMPLETE
0x80004001	DSERR_UNSUPPORTED
0x80004005	DSERR_GENERIC
0x80070005	DSERR_ACCESSDENIED
0x80070057	DSERR_INVALIDPARAM
0x8878000A	DSERR_ALLOCATED
0x8878001E	DSERR_CONTROLUNAVAIL
0x88780032	DSERR_INVALIDCALL
0x88780046	DSERR_PRIOLEVELNEEDED
0x88780064	DSERR_BADFORMAT
0x88780078	DSERR_NODRIVER
0x88780082	DSERR_ALREADYINITIALIZED
0x88780096	DSERR_BUFFERLOST
0x887800A0	DSERR_OTHERAPPHASPRIO
0x887800AA	DSERR_UNINITIALIZED
0x887810B4	DSERR_BUFFERTOOSMALL
0x887810BE	DSERR_DS8_REQUIRED
0x88781161	DSERR_OBJECTNOTFOUND

DirectSound Return Values by Name

The following list describes all DirectSound return values. To find a constant from its value, see DirectSound Return Values By Number.

DS_OK
: The method succeeded.

DS_NO_VIRTUALIZATION
: The buffer was created, but another 3-D algorithm was substituted.

DS_INCOMPLETE
: The method succeeded, but not all the optional effects were obtained.

DSERR_ACCESSDENIED
: The request failed because access was denied.

DSERR_ALLOCATED
: The request failed because resources, such as a priority level, were already in use by another caller.

DSERR_ALREADYINITIALIZED

The object is already initialized.

DSERR_BADFORMAT
The specified wave format is not supported.

DSERR_BUFFERLOST
The buffer memory has been lost and must be restored.

DSERR_BUFFERTOOSMALL
The buffer size is not great enough to enable effects processing.

DSERR_CONTROLUNAVAIL
The buffer control (volume, pan, and so on) requested by the caller is not available.

DSERR_DS8_REQUIRED
A DirectSound object of class CLSID_DirectSound8 or later is required for the requested functionality. For more information, see **IDirectSound8**.

DSERR_GENERIC
An undetermined error occurred inside the DirectSound subsystem.

DSERR_INVALIDCALL
This function is not valid for the current state of this object.

DSERR_INVALIDPARAM
An invalid parameter was passed to the returning function.

DSERR_NOAGGREGATION
The object does not support aggregation.

DSERR_NODRIVER
No sound driver is available for use, or the given GUID is not a valid DirectSound device ID.

DSERR_NOINTERFACE
The requested COM interface is not available.

DSERR_OBJECTNOTFOUND
The requested object was not found.

DSERR_OTHERAPPHASPRIO
Another application has a higher priority level, preventing this call from succeeding.

DSERR_OUTOFMEMORY
The DirectSound subsystem could not allocate sufficient memory to complete the caller's request.

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

DSERR_UNINITIALIZED
The **IDirectSound8::Initialize** method has not been called or has not been called successfully before other methods were called.

DSERR_UNSUPPORTED
The function called is not supported at this time.

DirectMusic Visual Basic Reference

This section contains reference information for the API elements of Microsoft® DirectMusic® for Microsoft® Visual Basic®. Reference material is divided into the following categories:

- DirectMusic Classes
- DirectMusic Types
- DirectMusic Enumerations
- DirectMusic Error Codes

DirectMusic Classes

This section contains references for methods of the following DirectMusic classes.

- **DirectMusicAudioPath8**
- **DirectMusicBand8**
- **DirectMusicChordMap8**
- **DirectMusicCollection8**
- **DirectMusicComposer8**
- **DirectMusicLoader8**
- **DirectMusicPerformance8**
- **DirectMusicSegment8**
- **DirectMusicSegmentState8**
- **DirectMusicStyle8**

DirectMusicAudioPath8

[#]Represents the stages of data flow from the performance to the DirectSound buffer. A standard audiopath can be created by using **DirectMusicPerformance8.CreateStandardAudioPath**. A standard default path can also be created by **DirectMusicPerformance8.InitAudio** and retrieved by using **DirectMusicPerformance8.GetDefaultAudioPath**. Finally, an audiopath can be created from an audiopath configuration in a segment by using the **DirectMusicPerformance8.CreateAudioPath** method.

A **DirectMusicAudioPath8** object is passed to **DirectMusicPerformance8.PlaySegmentEx** to play the segment on that audiopath.

The **DirectMusicAudioPath8** class has the following methods.

IDH_DirectMusicAudioPath8_dxaudio_vb

DirectMusicAudioPath8.Activate

#Activates or deactivates the audiopath.

object.**Activate(***fActive* **As Boolean)**

Parts

object
Resolves to a **DirectMusicAudioPath8** object.

fActive
Boolean value that specifies the state of the path. A value of True activates the path, and False deactivates it.

Error Codes

If the method fails, an error is raised.

DirectMusicAudioPath8.GetObjectInPath

#Retrieves an object in the audiopath.

```
object.GetObjectInPath( _
  lPChannel As Long, _
  lStage As CONST_DMUSIC_PATH, _
  lBuffer As Long,
  guidObject As String, _
  lIndex As Long, _
  iidInterface As String _
) As Unknown
```

Parts

object
Resolves to a **DirectMusicAudioPath8** object.

lPChannel
Performance channel to search, or DMUS_PCHANNEL_ALL to search all channels. The first channel is numbered 0. See Remarks.

lStage

IDH_DirectMusicAudioPath8.Activate_dxaudio_vb
IDH_DirectMusicAudioPath8.GetObjectInPath_dxaudio_vb

Stage in the path. One of the constants from the **CONST_DMUSIC_PATH** enumeration. See Remarks.

lBuffer

If *lStage* is DMUS_PATH_BUFFER_DMO or DMUS_PATH_MIXIN_BUFFER_DMO, specifies the index of the buffer in which that DMO resides. If *lStage* is DMUS_PATH_BUFFER or DMUS_PATH_MIXIN_BUFFER, specifies the index of the buffer. Otherwise must be 0.

guidObject

Class identifier of the object, such as CLSID_DirectMusicPerformance, or GUID_ALL_OBJECTS to search for an object of any class. This parameter is ignored if only a single class of object can exist at the stage specified by *lStage*, and can be set to GUID_NULL.

lIndex

Index of the object within a list of matching objects. Set to 0 to find the first matching object. If *lStage* is DMUS_PATH_BUFFER or DMUS_PATH_MIXIN_BUFFER, this parameter is ignored, and the buffer index is specified by *lBuffer*.

iidInterface

Identifier of the desired class. This can be a GUID in string form, or one of the identifiers in the AUDIOSTRINGCONSTANTS module, such as IID_DirectSoundPrimaryBuffer.

Return Values

Returns an object of the class specified in *iidInterface*.

Error Codes

If the method fails, an error is raised. Possible values of Err.Number include the following:

DMUS_E_NOT_FOUND

DMUS_E_NOINTERFACE

Remarks

The value in *lPChannel* must be 0 for any stage that is not channel-specific. Objects in the following stages are channel-specific and can be retrieved by setting a channel number or DMUS_PCHANNEL_ALL in *lPChannel*:

DMUS_PATH_AUDIOPATH_TOOL
DMUS_PATH_BUFFER
DMUS_PATH_BUFFER_DMO
DMUS_PATH_PERFORMANCE_TOOL

DMUS_PATH_PORT
DMUS_PATH_SEGMENT_TOOL

The precedence of the parameters in filtering out unwanted objects is as follows:

1. *lStage*.
2. *guidObject*. If this value is not GUID_All_Objects, only objects whose class identifier equals *guidObject* are searched. However, this parameter is ignored if only a single class of object can be found at the specified stage.
3. *lPChannel*. If the stage is channel-specific and this value is not DMUS_PCHANNEL_ALL, only objects on the channel are searched.
4. *lBuffer*. This is used only if *lStage* is DMUS_PATH_BUFFER, DMUS_PATH_MIXIN_BUFFER, DMUS_PATH_BUFFER_DMO, or DMUS_PATH_MIXIN_BUFFER_DMO.
5. *lIndex*.

If a matching object is found but the class specified by *iidInterface* cannot be obtained, the method fails.

Not all types of objects that can be retrieved by this method are supported in DirectX for Visual Basic. For example, it would be meaningless to pass DMUS_PATH_PERFORMANCE_TOOL in *lStage*, because tools are not supported.

See Also

Retrieving Objects from an Audiopath

DirectMusicAudioPath8.SetVolume

[#]Sets the audio volume on the path.

object.SetVolume(*lVolume* **As Long,** *lDuration* **As Long)**

Parts

object
Resolves to a **DirectMusicAudioPath8** object.

lVolume
Value that specifies the attenuation, in hundredths of a decibel.

lDuration
Value that specifies the time, in milliseconds, over which the volume change takes place. A value of 0 ensures maximum efficiency.

IDH_DirectMusicAudioPath8.SetVolume_dxaudio_vb

Error Codes

If the method fails, an error is raised.

DirectMusicBand8

#Represents a band, which is used to set the instrument choices and mixer settings for a set of performance channels.

Bands can be stored directly in their own files or embedded in a style. The **DirectMusicBand8** object is obtained by using one of the following methods:

- **DirectMusicLoader8.LoadBand**
- **DirectMusicLoader8.LoadBandFromResource**
- **DirectMusicStyle8.GetDefaultBand**
- **DirectMusicStyle8.GetBand**

The **DirectMusicBand8** class has the following methods:

Segment creation	**CreateSegment**
Instrument data	**Download**
	Unload

DirectMusicBand8.CreateSegment

#Creates a segment object that can be used to perform the volume, pan, transposition, and patch change commands in the band dynamically, using the **DirectMusicPerformance8.PlaySegmentEx** method.

object.**CreateSegment() As DirectMusicSegment8**

Parts

object
Resolves to a **DirectMusicBand8** object.

Return Values

Returns a **DirectMusicSegment8** object.

Error Codes

If the method fails, **Err.Number** can be set to one of the following values:

DMUS_E_FAIL

IDH_DirectMusicBand8_dxaudio_vb
IDH_DirectMusicBand8.CreateSegment_dxaudio_vb

DMUS_E_OUTOFMEMORY

DirectMusicBand8.Download

#Downloads the DLS data for instruments in the band to a performance object. Once a band has been downloaded, the instruments in the band can be selected, either individually with program change MIDI messages, or all at once by playing a band segment created through a call to the **DirectMusicBand8.CreateSegment** method.

object.**Download(***downloadpath* **As Unknown)**

Parts

object
Resolves to a **DirectMusicBand8** object.

downloadpath
DirectMusicAudioPath8 or **DirectMusicPerformance8** object that specifies the path on which the band is to perform.

Error Codes

If the method fails, and **Err.Number** can be set to one of the following values:

DMUS_E_NOT_INIT
DMUS_E_OUTOFMEMORY

Remarks

Because a downloaded band uses synthesizer resources, it should be unloaded when no longer needed by using the **DirectMusicBand8.Unload** method.

If DMUS_E_NOT_INIT is raised, it usually means that the performance was not properly connected to an initialized port. Since this is a complete failure, there is no need to call **DirectMusicBand8.Unload** later.

If the download only partially succeeds, no error is raised but some instruments might not play. The following are some common causes of a partial download:

- The band has instruments on performance channels not allocated on the port.
- The band has instruments in a DLS format incompatible with the synthesizer they are being downloaded to.

IDH_DirectMusicBand8.Download_dxaudio_vb

DirectMusicBand8.Unload

#Unloads the DLS data for instruments in the band previously downloaded by **DirectMusicBand8.Download**.

object.**Unload(***downloadpath* **As Unknown)**

Parts

object
Resolves to a **DirectMusicBand8** object.

downloadpath
DirectMusicAudioPath8 or **DirectMusicPerformance8** object that specifies the path to which the band was downloaded.

Error Codes

If the method fails, it raises an error. Possible values of **Err.Number** include the following:

DMUS_E_NOT_INIT
DMUS_E_NOT_FOUND

DirectMusicChordMap8

#Represents a chordmap. Chordmaps provide the **DirectMusicComposer8** object with the information it needs to create chord progressions for segments it composes. A chordmap can also be applied to an existing segment in order to change the chords.

The class has no public methods. An instance of it is obtained by using the **DirectMusicPerformance8.GetChordmap** method.

See Also

DirectMusicComposer8.AutoTransition,
DirectMusicComposer8.ChangeChordMap,
DirectMusicComposer8.ComposeSegmentFromShape,
DirectMusicComposer8.ComposeSegmentFromTemplate,
DirectMusicComposer8.ComposeTransition,
DirectMusicPerformance8.GetChordmap

IDH_DirectMusicBand8.Unload_dxaudio_vb
IDH_DirectMusicChordMap8_dxaudio_vb

DirectMusicCollection8

#Manages an instance of a DLS file.

The class has no public methods. An instance of it is obtained by using the **DirectMusicLoader8.LoadCollection** or the **DirectMusicLoader8.LoadCollectionFromResource** method, and is associated with a segment by a call to **DirectMusicSegment8.ConnectToCollection**.

DirectMusicComposer8

#Used to compose segments and transitions from compositional elements, and to change the chordmap of an existing segment.

A **DirectMusicComposer8** object is obtained by using the **DirectX8.DirectMusicComposerCreate** method.

The methods of the **DirectMusicComposer8** class can be organized into the following groups.

Changing chordmaps	**ChangeChordMap**
Composing ordinary segments	**ComposeSegmentFromShape**
	ComposeSegmentFromTemplate
Composing template segments	**ComposeTemplateFromShape**
Composing transition segments	**AutoTransition**
	ComposeTransition

DirectMusicComposer8.AutoTransition

#Composes a transition from inside a performance's primary segment (or from silence) to another segment, and then cues the transition and the second segment to play.

```
object.AutoTransition( _
   performance As DirectMusicPerformance8, _
   toSeg As DirectMusicSegment8, _
   lCommand As Long, _
   lFlags As Long, _
   chordmap As DirectMusicChordMap8 _
) As DirectMusicSegment8
```

IDH_DirectMusicCollection8_dxaudio_vb
IDH_DirectMusicComposer8_dxaudio_vb
IDH_DirectMusicComposer8.AutoTransition_dxaudio_vb

Parts

object
Resolves to a **DirectMusicComposer8** object.

performance
DirectMusicPerformance8 object in which to do the transition.

toSeg
Segment to which the transition should smoothly flow. See Remarks.

lCommand
Embellishment to use when composing the transition. See **CONST_DMUS_COMMANDT_TYPES**. If this value is DMUS_COMMANDT_ENDANDINTRO, the method composes a segment containing both an ending to the primary segment and an introduction to *toSeg*.

lFlags
Composition options. See **CONST_DMUS_COMPOSEF_FLAGS**.

chordmap
DirectMusicChordMap8 to be used when composing the transition.

Return Values

Returns a **DirectMusicSegment8** object, unless no style is available for the composition of the transitional segment. See Remarks.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The value in *toSeg* can be Nothing, as long as *lFlags* does not include DMUS_COMPOSEF_MODULATE. If *toSeg* is Nothing or a segment that contains no style track, intro embellishments are not valid. If there is no currently playing segment or it contains no style track, then fill, break, end, and groove embellishments are not valid.

It is possible for both the currently playing segment and *segment1* to be Nothing or segments that contain no style tracks (such as segments based on MIDI files). If so, all embellishments are invalid, and no transition occurs between the currently playing segment and *segment1*. The method returns Nothing, but it succeeds and cues the segment represented by *segment1*.

The value in *chordmap* can be Nothing. If so, an attempt is made to obtain a chordmap from a chordmap track, first from *segment1* and then from the performance's primary segment. If neither of these segments contains a chordmap track, the chord occurring at the current time in the primary segment is used as the chord in the transition.

DirectMusicComposer8.ChangeChordMap

#The **DirectMusicComposer8.ChangeChordMap** method modifies the chords and scale pattern of an existing segment to reflect a new chordmap.

object.**ChangeChordMap(***segment* **As DirectMusicSegment8, _**
trackScale **As Boolean, _**
chordmap **As DirectMusicChordMap8)**

Parts

object
Resolves to a **DirectMusicComposer8** object.

segment
DirectMusicSegment8 object that specifies the segment in which to change the chordmap.

trackScale
Value of type **Boolean** that specifies scale tracking. If True, the method transposes all the chords to be relative to the root of the new chordmap's scale. If False, roots are left as they were.

chordmap
DirectMusicChordMap8 object that specifies the new chordmap for the segment.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The method can be called while the segment is playing.

DirectMusicComposer8.ComposeSegmentFromShape

#Creates an original segment from a style and a chordmap, based on a predefined shape. The shape represents the way chords and embellishments occur over time across the segment.

object.**ComposeSegmentFromShape(***style* **As DirectMusicStyle8, _**
numberOfMeasures **As Integer, _**

IDH_DirectMusicComposer8.ChangeChordMap_dxaudio_vb
IDH_DirectMusicComposer8.ComposeSegmentFromShape_dxaudio_vb

```
    shape As Integer, _
    activity As Integer, _
    bIntro As Boolean, _
    bEnd As Boolean, _
    chordmap As DirectMusicChordMap8 _
) As DirectMusicSegment8
```

Parts

object
Resolves to a **DirectMusicComposer8** object.

style
DirectMusicStyle8 object that specifies the style from which to compose the segment.

numberOfMeasures
Length, in measures, of the segment to be composed.

shape
Shape of the segment to be composed, based on changes in the groove level. Possible values are of the **CONST_DMUS_SHAPET_TYPES** enumeration.

activity
Rate of harmonic motion. Valid values are from 0 through 3. Lower values mean more chord changes.

bIntro
True if an introduction is to be composed for the segment, False otherwise.

bEnd
True if an ending is to be composed for the segment, False otherwise.

chordmap
DirectMusicChordMap8 object from which to create the segment.

Return Values

Returns a **DirectMusicSegment8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicComposer8.ComposeSegmentFromTemplate, **DirectMusicComposer8.ComposeTemplateFromShape**

DirectMusicComposer8.ComposeSegmentFromTemplate

[#]Creates an original segment from a style, a chordmap, and a template.

```
object.ComposeSegmentFromTemplate( _
    style As DirectMusicStyle8, _
    templateSeg As DirectMusicSegment8, _
    activity As Integer, _
    chordmap As DirectMusicChordMap8 _
) As DirectMusicSegment8
```

Parts

object
Resolves to a **DirectMusicComposer8** object.

style
DirectMusicStyle8 object from which to create the segment.

templateSeg
DirectMusicSegment8 object representing the template from which to create the segment.

activity
Rate of harmonic motion. Valid values are from 0 through 3. Lower values mean more chord changes.

chordmap
DirectMusicChordMap8 object representing the chordmap from which to create the segment.

Return Values

Returns a **DirectMusicSegment8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_INVALIDARG

Remarks

If s*tyle* is not Nothing, it is used in composing the segment; if it is Nothing, a style is retrieved from the template specified in *templateSeg*. Similarly, if *chordmap* is not

---

[#] IDH_DirectMusicComposer8.ComposeSegmentFromTemplate_dxaudio_vb

Nothing, it is used in composing the segment; if it is Nothing, a chordmap is retrieved from the template.

If *style* is Nothing and there is no style track in the template, or *chordmap* is Nothing and there is no chordmap track, the method returns DMUS_E_INVALIDARG.

The length of the segment is equal to the length of the template passed in.

See Also

DirectMusicComposer8.ComposeSegmentFromShape,
DirectMusicComposer8.ComposeTemplateFromShape

DirectMusicComposer8.ComposeTemplateFromShape

[#]Creates a new template segment based on a predefined shape.

object.**ComposeTemplateFromShape(***numMeasures* **As Integer, _**
shape **As Integer, _**
bIntro **As Boolean, _**
bEnd **As Boolean, _**
endLength **As Integer _**
) As DirectMusicSegment8

Parts

object
Resolves to a **DirectMusicComposer8** object.

numMeasures
Length, in measures, of the segment to be composed. This value must be greater than 0.

shape
Shape of the segment to be composed, based on groove levels. Possible values are of the **CONST_DMUS_SHAPET_TYPES** enumeration.

bIntro
True if an introduction is to be composed for the segment, False otherwise.

bEnd
True if an ending is to be composed for the segment, False otherwise.

endLength
Length in measures of the ending, if one is to be composed. If *bEnd* is True, this value must be greater than 0 and equal to or less than the number of measures available (that is, not used in the introduction). See also Remarks.

IDH_DirectMusicComposer8.ComposeTemplateFromShape_dxaudio_vb

Return Values

Returns a **DirectMusicSegment8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY

Remarks

The value of *endLength* should not be greater than the length of the longest ending available in any style likely to be associated with this template through the **DirectMusicComposer8.ComposeSegmentFromTemplate** method. The ending starts playing at *endLength* measures before the end of the segment. If the ending is less than *endLength* measures long, the music then reverts to the regular groove.

See Also

DirectMusicComposer8.ComposeSegmentFromShape, **DirectMusicComposer8.ComposeSegmentFromTemplate**

DirectMusicComposer8.ComposeTransition

#Composes a transition from a measure inside one segment to another.

```
object.ComposeTransition(fromSeg As DirectMusicSegment8, _
    toSeg As DirectMusicSegment8, _
    mtTime As Long, _
    lCommand As Long, _
    lFlags As Long, _
    chordmap As DirectMusicChordMap8 _
) As DirectMusicSegment8
```

Parts

object
Resolves to a **DirectMusicComposer8** object.

fromSeg
DirectMusicSegment8 object representing the segment from which to compose the transition.

IDH_DirectMusicComposer8.ComposeTransition_dxaudio_vb

toSeg
Segment to which the transition should smoothly flow. Can be Nothing if *lFlags* does not include DMUS_COMPOSEF_MODULATE.

mtTime
Time in *fromSeg* from which to compose the transition.

lCommand
Embellishment to use when composing the transition. See **CONST_DMUS_COMMANDT_TYPES**. If this value is DMUS_COMMANDT_ENDANDINTRO, the method composes a segment containing both an ending to *fromSeg* and an introduction to *toSeg*.

lFlags
Composition options. This parameter can contain one or more of the **CONST_DMUS_COMPOSEF_FLAGS** enumeration.

chordmap
DirectMusicChordMap8 object representing the chordmap to be used when composing the transition. See Remarks.

Return Values

Returns a **DirectMusicSegment8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY

Remarks

The value in *chordmap* can be Nothing. If so, an attempt is made to obtain a chordmap from a chordmap track, first from *toSeg* and then from *fromSeg*. If neither of these segments contains a chordmap track, the chord occurring at *mtTime* in *fromSeg* is used as the chord in the transition.

The composer looks for a tempo, first in *fromSeg*, and then in *toSeg*. If neither of those segments contains a tempo track, the tempo for the transition segment is taken from the style.

See Also

DirectMusicComposer8.AutoTransition

DirectMusicLoader8

#Used to find and load objects that represent musical and instrument data.

The object is obtained by using the **DirectX8.DirectMusicLoaderCreate** method.

Objects referred to by other objects are loaded automatically. For example, a style may contain references to bands and collections in other files, and these are loaded, if possible, when the **DirectMusicLoader8.LoadStyle** method is called.

The methods of **DirectMusicLoader8** can be organized into the following groups:

Loading	**LoadBand**
	LoadBandFromResource
	LoadChordMap
	LoadChordMapFromResource
	LoadCollection
	LoadCollectionFromResource
	LoadSegment
	LoadSegmentFromResource
	LoadStyle
	LoadStyleFromResource
Searching	**SetSearchDirectory**

DirectMusicLoader8.LoadBand

#Loads a band from a file.

object.**LoadBand(***filename* **As String) As DirectMusicBand8**

Parts

object
Resolves to a **DirectMusicLoader8** object.

filename
Name of the file containing the band. If the file is not in the current directory or in the directory set by **DirectMusicLoader8.SetSearchDirectory**, the full path must be given.

Return Values

Returns a **DirectMusicBand8** object.

IDH_DirectMusicLoader8_dxaudio_vb
IDH_DirectMusicLoader8.LoadBand_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_INVALID_BAND
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader8.LoadBandFromResource

DirectMusicLoader8.LoadBandFromResource

[#]Loads a band from a resource.

```
object.LoadBandFromResource( _
    moduleName As String, _
    resourceName As String _
) As DirectMusicBand8
```

Parts

object
Resolves to a **DirectMusicLoader8** object.

moduleName
Name of the module containing the resource.

resourceName
Identifier of the resource containing the band. The resource must be of type "DMBAND".

Return Values

Returns a **DirectMusicBand8** object.

IDH_DirectMusicLoader8.LoadBandFromResource_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_INVALID_BAND
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader8.LoadBand

DirectMusicLoader8.LoadChordMap

#Loads a chordmap from a file.

object.**LoadChordMap(***filename* **As String _**
) As DirectMusicChordMap8

Parts

object
Resolves to a **DirectMusicLoader8** object.

filename
Name of the file containing the chordmap. If the file is not in the current directory or in the directory set by **DirectMusicLoader8.SetSearchDirectory**, the full path must be given.

Return Values

Returns a **DirectMusicChordMap8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_CHUNKNOTFOUND
DMUS_E_FAIL
DMUS_E_INVALIDARG

IDH_DirectMusicLoader8.LoadChordMap_dxaudio_vb

DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader8.LoadChordMapFromResource

DirectMusicLoader8.LoadChordMapFromResource

#Loads a chordmap from a resource.

object.**LoadChordMapFromResource(_**
moduleName **As String, _**
resourceName **As String _**
) As DirectMusicChordMap8

Parts

object
Resolves to a **DirectMusicLoader8** object.

moduleName
Name of the module containing the resource.

resourceName
Identifier of the resource containing the chordmap. The resource must be of type "DMCHORD".

Return Values

Returns a **DirectMusicChordMap8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_CHUNKNOTFOUND
DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE

IDH_DirectMusicLoader8.LoadChordMapFromResource_dxaudio_vb

DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader8.LoadChordMap

DirectMusicLoader8.LoadCollection

#Loads a DLS collection from a file.

object.**LoadCollection(***filename* **As String _**
) As DirectMusicCollection8

Parts

object
Resolves to a **DirectMusicLoader8** object.

filename
Name of the file containing the DLS collection. If the file is not in the current directory or in the directory set by **DirectMusicLoader8.SetSearchDirectory**, the full path must be given.

Return Values

Returns a **DirectMusicCollection8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_CHUNKNOTFOUND
DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED
DMUS_E_NOTADLSCOL

See Also

DirectMusicLoader8.LoadCollectionFromResource

IDH_DirectMusicLoader8.LoadCollection_dxaudio_vb

DirectMusicLoader8.LoadCollectionFromResource

[#]Loads a DLS collection from a resource.

```
object.LoadCollectionFromResource( _
  moduleName As String, _
  resourceName As String _
) As DirectMusicCollection8
```

Parts

object
Resolves to a **DirectMusicLoader8** object.

moduleName
Name of the module containing the resource.

resourceName
Identifier of the resource containing the collection. The resource must be of type "DMCOLL".

Return Values

Returns a **DirectMusicCollection8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_CHUNKNOTFOUND
DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED
DMUS_E_NOTADLSCOL

See Also

DirectMusicLoader8.LoadCollection

IDH_DirectMusicLoader8.LoadCollectionFromResource_dxaudio_vb

DirectMusicLoader8.LoadSegment

[#]Loads a segment from a MIDI file, a wave file, or a segment file authored in DirectMusic Producer.

object.**LoadSegment(***filename* **As String _**
) As DirectMusicSegment8

Parts

object
Resolves to a **DirectMusicLoader8** object.

filename
Name of the file containing the segment. If the file is not in the current directory or in the directory set by **DirectMusicLoader8.SetSearchDirectory**, the full path must be given.

Return Values

Returns a **DirectMusicSegment8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_CHUNKNOTFOUND
DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader8.LoadSegmentFromResource

[#] IDH_DirectMusicLoader8.LoadSegment_dxaudio_vb

DirectMusicLoader8.LoadSegmentFromResource

#Loads a segment from a resource. The segment can be in the format of a MIDI file, a wave file, or a segment authored in DirectMusic Producer.

```
object.LoadSegmentFromResource( _
    moduleName As String, _
    resourceName As String _
) As DirectMusicSegment8
```

Parts

object
Resolves to a **DirectMusicLoader8** object.

moduleName
Name of the module containing the resource. The resource must be of type "DMSEG".

resourceName
Identifier of the resource containing the segment.

Return Values

Returns a **DirectMusicSegment8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_CHUNKNOTFOUND
DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED
DMUS_E_UNSUPPORTED_STREAM

See Also

DirectMusicLoader8.LoadSegment

IDH_DirectMusicLoader8.LoadSegmentFromResource_dxaudio_vb

DirectMusicLoader8.LoadStyle

#Loads a style object from a file.

object.**LoadStyle(***filename* **As String) As DirectMusicStyle8**

Parts

object
Resolves to a **DirectMusicLoader8** object.

filename
Name of the file containing the style object. If the file is not in the current directory or in the directory set by **DirectMusicLoader8.SetSearchDirectory**, the full path must be given.

Return Values

Returns a **DirectMusicStyle8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_CHUNKNOTFOUND
DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader8.LoadStyleFromResource

DirectMusicLoader8.LoadStyleFromResource

#Loads a style object from a resource.

object.**LoadStyleFromResource(_**

IDH_DirectMusicLoader8.LoadStyle_dxaudio_vb
IDH_DirectMusicLoader8.LoadStyleFromResource_dxaudio_vb

moduleName **As String, _**
resourceName **As String _**
) As DirectMusicStyle8

Parts

object
Resolves to a **DirectMusicLoader8** object.

moduleName
Name of the module containing the resource.

resourceName
Identifier of the resource containing the style object. The resource must be of type "DMSTYLE".

Return Values

Returns a **DirectMusicStyle8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_CHUNKNOTFOUND
DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader8.LoadStyle

DirectMusicLoader8.SetSearchDirectory

#Sets the directory to be searched by the **DirectMusicLoader8.LoadBand**, **DirectMusicLoader8.LoadChordmap**, **DirectMusicLoader8.LoadCollection**, **DirectMusicLoader8.LoadSegment**, and **DirectMusicLoader8.LoadStyle** methods when a fully qualified path is not supplied.

object.**SetSearchDirectory(***dir* **As String)**

IDH_DirectMusicLoader8.SetSearchDirectory_dxaudio_vb

Parts

object
Resolves to a **DirectMusicLoader8** object.

dir
Directory to search.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_BADPATH

Remarks

Once a search path is set, the loader does not need a full path every time it is given an object to load by file name. This allows objects that refer to other objects to find them by file name without knowing the full path.

DirectMusicPerformance8

#Overall manager of music playback. The performance maps performance channels to audiopaths, plays segments, dispatches messages, requests and receives event notifications, and sets and retrieves music parameters. It also has several methods for getting information about timing and for converting time and music values from one system to another.

If an application wants to have two or more complete sets of music playing at the same time, it can do so by creating more than one performance object. Separate performances obey separate tempo maps and play completely asynchronously, whereas all segments within one performance play in lock step.

The **DirectMusicPerformance8** object is obtained by using the **DirectX8.DirectMusicPerformanceCreate** method.

The methods of the **DirectMusicPerformance8** class can be organized into the following groups.

Audiopaths	**CreateAudioPath**
	CreateStandardAudioPath
	GetDefaultAudioPath

IDH_DirectMusicPerformance8_dxaudio_vb

	SetDefaultAudioPath
Initialization and Cleanup	**InitAudio**
	CloseDown
Messages	**SendCurvePMSG**
	SendMIDIPMSG
	SendNotePMSG
	SendPatchPMSG
	SendTempoPMSG
	SendTimeSigPMSG
	SendTransposePMSG
Notification	**AddNotificationType**
	GetNotificationPMSG
	RemoveNotificationType
	SetNotificationHandle
Parameters	**GetChordmap**
	GetCommand
	GetGrooveLevel
	GetMasterAutoDownload
	GetMasterGrooveLevel
	GetMasterTempo
	GetMasterVolume
	GetStyle
	GetTempo
	GetTimeSig
	Reset
	SetMasterAutoDownload
	SetMasterGrooveLevel
	SetMasterTempo
	SetMasterVolume
Segments	**GetSegmentState**
	IsPlaying
	PlaySegmentEx
	StopEx
Timing	**AdjustTime**
	ClockToMusicTime
	GetBumperLength
	GetClockTime

	GetLatencyTime
	GetMusicTime
	GetPrepareTime
	GetQueueTime
	GetResolvedTime
	MusicToClockTime
	SetBumperLength
	SetPrepareTime
Miscellaneous	**Invalidate**

DirectMusicPerformance8.AddNotificationType

[#]Causes the performance to generate notification messages whenever events of the requested type occur.

object.**AddNotificationType(** _
type **As CONST_DMUS_NOTIFICATION_TYPE)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

type
Type of event for which notification messages are to be sent. For possible values, see **CONST_DMUS_NOTIFICATION_TYPE**.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY

See Also

DirectMusicPerformance8.RemoveNotificationType

[#] IDH_DirectMusicPerformance8.AddNotificationType_dxaudio_vb

DirectMusicPerformance8.AdjustTime

#Adjusts the internal performance time forward or backward. This is mostly used to compensate for drift when synchronizing to another source.

object.**AdjustTime(***tAmount* **As Long)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

tAmount
Amount of time, in clock time units, to add or subtract. This can be a number in the range from –1000 through 1000 (–1 second through +1 second).

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_INVALIDARG.

Remarks

The adjusted time is used internally by DirectMusic. It is not reflected in the time retrieved by the **DirectMusicPerformance8.GetClockTime** method.

DirectMusicPerformance8.ClockToMusicTime

#Converts clock time to music time.

object.**ClockToMusicTime(***ctTime* **As Long) As Long**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

ctTime
Time to convert, in clock time units.

Return Values

Returns the equivalent music time.

IDH_DirectMusicPerformance8.AdjustTime_dxaudio_vb
IDH_DirectMusicPerformance8.ClockToMusicTime_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

Remarks

This method converts a time offset from the start of the performance, not a duration. The ratio between music time and clock time units depends on the tempo, so in making the conversion, DirectMusic takes into account all tempo changes since the start of the performance. If a master tempo has been set for the performance, it is taken into account as well.

Because music time is less precise than clock time, rounding off occurs.

See Also

DirectMusicPerformance8.MusicToClockTime, Clock Time vs. Music Time

DirectMusicPerformance8.CloseDown

[#]Closes down the performance. An application that created the performance object and called **DirectMusicPerformance8.InitAudio** on it must call **CloseDown** before the performance object is released.

object.**CloseDown()**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

Failure to call **CloseDown** can cause memory leaks or application failures.

If the **DirectSound8** object was created in the call to **DirectMusicPerformance8.InitAudio** but no object was returned to the application, **CloseDown** also releases DirectSound and all DirectSound buffers. If your application has set any variables to DirectSound buffer objects, it should ensure that those variables go out of scope or are set to Nothing before calling **Closedown**.

If the application created **DirectSound8** explicitly, or obtained the **DirectSound8** object from **InitAudio**, it is responsible for releasing DirectSound.

IDH_DirectMusicPerformance8.CloseDown_dxaudio_vb

CloseDown releases any downloaded instruments that have not been unloaded.

DirectMusicPerformance8.CreateAudioPath

#Creates an audiopath object that represents the stages in data flow from the performance to DirectSound buffers.

```
object.CreateAudioPath( _
  SourceConfig As Unknown _
) As DirectMusicAudioPath8
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

SourceConfig
Object that represents the audiopath configuration. This object is typically obtained from a segment by using the **DirectMusicSegment8.GetAudioPathConfig** method.

Return Values

Returns a **DirectMusicAudioPath8** object.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include the following:

DMUS_E_NOT_INIT
DMUS_E_AUDIOPATHS_NOT_VALID
DMUS_E_OUTOFMEMORY

DirectMusicPerformance8.CreateStandardAudioPath

#Creates an object that represents the stages in data flow from the performance to DirectSound buffers. This method can be used instead of **DirectMusicPerformance8.CreateAudioPath** to create a basic predefined audiopath rather than one defined in a source file.

```
object.CreateStandardAudioPath( _
```

IDH_DirectMusicPerformance8.CreateAudioPath_dxaudio_vb
IDH_DirectMusicPerformance8.CreateStandardAudioPath_dxaudio_vb

lType **As CONST_DMUSIC_STANDARD_AUDIO_PATH, _**
lPChannelCount **As Long, _**
fActive **As Boolean _**
) As DirectMusicAudioPath8

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lType
Type of the path, specified by one of the constants of the **CONST_DMUSIC_STANDARD_AUDIO_PATH** enumeration. For more information on these audiopath types, see Standard Audiopaths.

lPChannelCount
Value that specifies the number of performance channels in the path.

fActive
Boolean value that specifies whether to activate the path on creation.

Return Values

Returns a **DirectMusicAudioPath8** object.

Error Codes

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
DMUS_E_AUDIOPATHS_NOT_VALID
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY

DirectMusicPerformance8.GetBumperLength

[#]Retrieves the amount of time between the time at which messages are placed in the port buffer and the time at which they begin to be processed by the port.

object.**GetBumperLength() As Long**

Parts

object

IDH_DirectMusicPerformance8.GetBumperLength_dxaudio_vb

Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns the bumper length, in milliseconds.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The default value is 50 milliseconds.

See Also

DirectMusicPerformance8.SetBumperLength

DirectMusicPerformance8.GetChordmap

[#]Retrieves the chordmap from the performance's control segment.

```
object.GetChordMap( _
  mtTime As Long, _
  mtUntil As Long _
) As DirectMusicChordMap8
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

mtTime
Time for which the chordmap is to be retrieved, in music time.

mtUntil
Variable of type **Long** that receives the music time (relative to *mtTime*) until which the chordmap is valid. If this returns a value of 0, either the chordmap is always valid, or it is unknown when it might become invalid. See Remarks.

Return Values

Returns a **DirectMusicChordMap8** object.

IDH_DirectMusicPerformance8.GetChordmap_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

Remarks

Normally the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. The object returned by the method can become invalid before the time returned in *mtUntil* if another control segment is cued.

See Also

CONST_DMUS_SEGF_FLAGS

DirectMusicPerformance8.GetClockTime

#Retrieves the current time.

object.**GetClockTime() As Long**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns the time in units of approximately 1 millisecond, relative to an arbitrary start time.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

See Also

DirectMusicPerformance8.GetMusicTime, Clock Time vs. Music Time

---

IDH_DirectMusicPerformance8.GetClockTime_dxaudio_vb

DirectMusicPerformance8.GetCommand

#Retrieves a command from the performance's control segment. The command indicates what type of pattern is being played at the specified time.

```
object.GetCommand( _
  mtTime As Long, _
  mtUntil As Long _
) As Byte
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

mtTime
Time for which the command is to be retrieved, in music time.

mtUntil
Variable to receive the music time (relative to *mtTime*) until which the command is valid. If this returns a value of 0, either the command is always valid, or it is unknown when it might become invalid. See Remarks.

Return Values

Returns a command type. See **CONST_DMUS_COMMANDT_TYPES**.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following error codes:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

Remarks

Normally the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. The object returned by the method can become invalid before the time returned in *mtUntil* if another control segment is cued.

IDH_DirectMusicPerformance8.GetCommand_dxaudio_vb

See Also

CONST_DMUS_SEGF_FLAGS

DirectMusicPerformance8.GetDefaultAudioPath

#Retrieves the default audiopath set by **DirectMusicPerformance8.InitAudio** or **DirectMusicPerformance8.SetDefaultAudioPath**.

object.**GetDefaultAudioPath() As DirectMusicAudioPath8**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns a **DirectMusicAudioPath8** object.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include the following:

DMUS_E_AUDIOPATHS_NOT_VALID
DMUS_E_NOT_INIT

DirectMusicPerformance8.GetGrooveLevel

#Retrieves the groove level from the performance's control segment. The groove level determines which patterns can be played at the specified time.

```
object.GetGrooveLevel( _
  mtTime As Long, _
  mtUntil As Long _
) As Byte
```

Parts

object

IDH_DirectMusicPerformance8.GetDefaultAudioPath_dxaudio_vb
IDH_DirectMusicPerformance8.GetGrooveLevel_dxaudio_vb

Resolves to a **DirectMusicPerformance8** object.

mtTime

Time for which the groove level is to be retrieved, in music time.

mtUntil

Variable of type **Long** that receives the music time (relative to *mtTime*) until which the groove level is valid. If this returns a value of 0, either the groove level is always valid, or it is unknown when it might become invalid. See Remarks.

Return Values

Returns a value in the range from 1 through 100.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following error codes:

DMUS_E_NO_MASTER_CLOCK

DMUS_E_GET_UNSUPPORTED

DMUS_E_NOT_FOUND

DMUS_E_TRACK_NOT_FOUND

Remarks

Normally the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. The object returned by the method can become invalid before the time returned in *mtUntil* if another control segment is cued.

See Also

DirectMusicPerformance8.GetMasterGrooveLevel, **CONST_DMUS_SEGF_FLAGS**

DirectMusicPerformance8.GetLatencyTime

[#]Retrieves the current latency time. Latency time is the time at which messages are sent to the port to be rendered.

object.**GetLatencyTime() As Long**

IDH_DirectMusicPerformance8.GetLatencyTime_dxaudio_vb

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns the latency time, in clock time units.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK

DirectMusicPerformance8.GetMasterAutoDownload

[#]Retrieves the current setting for automatic downloading of instruments.

object.**GetMasterAutoDownload() As Boolean**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns True if autodownloading is on, and False otherwise. The default value is False.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicPerformance8.SetMasterAutoDownload, **DirectMusicSegment8.SetAutoDownloadEnable**

DirectMusicPerformance8.GetMasterGrooveLevel

[#] IDH_DirectMusicPerformance8.GetMasterAutoDownload_dxaudio_vb

#Retrieves the current master groove level, which is a value added to all groove levels in the performance. The resulting value is adjusted, if necessary, to fall within the range from 1 through 100.

object.**GetMasterGrooveLevel() As Integer**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns the master groove level.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicPerformance8.GetGrooveLevel,
DirectMusicPerformance8.SetMasterGrooveLevel

DirectMusicPerformance8.GetMasterTempo

#Retrieves the current master tempo.

object.**GetMasterTempo() As Single**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns a value in the range from 0.01 through 100.0.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

IDH_DirectMusicPerformance8.GetMasterGrooveLevel_dxaudio_vb
IDH_DirectMusicPerformance8.GetMasterTempo_dxaudio_vb

Remarks

The master tempo is a scaling factor that is applied to the tempo by the final output tool. By default it is 1. A value of 0.5 would halve the tempo, and a value of 2.0 would double it.

See Also

DirectMusicPerformance8.SetMasterTempo

DirectMusicPerformance8.GetMasterVolume

#Retrieves the current master volume.

object.**GetMasterVolume() As Long**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns the current master volume, in hundredths of a decibel.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The master volume is an amplification or attenuation factor applied to the default volume of the entire performance. The range of permitted values is determined by the port.

See Also

DirectMusicPerformance8.SetMasterVolume

DirectMusicPerformance8.GetMusicTime

#Retrieves the current time of the performance, in music time.

object.**GetMusicTime() As Long**

IDH_DirectMusicPerformance8.GetMasterVolume_dxaudio_vb
IDH_DirectMusicPerformance8.GetMusicTime_dxaudio_vb

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns the current time, in music time units.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

See Also

DirectMusicPerformance8.GetClockTime, Clock Time vs. Music Time

DirectMusicPerformance8.GetNotificationPMsg

[#]Retrieves a pending notification message.

```
object.GetNotificationPMSG( _
  message As DMUS_NOTIFICATION_PMSG _
) As Boolean
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

message
DMUS_NOTIFICATION_PMSG type that receives the message.

Return Values

Returns True if a message was received, and False if no message was pending.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

IDH_DirectMusicPerformance8.GetNotificationPMsg_dxaudio_vb

DirectMusicPerformance8.GetPrepareTime

#Retrieves the approximate interval between the time at which messages are prepared and the time at which they are processed and heard.

object.**GetPrepareTime() As Long**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Return Values

Returns the prepare time, in milliseconds.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The default value is 1000 milliseconds.

See Also

DirectMusicPerformance8.SetPrepareTime, Prepare Time

DirectMusicPerformance8.GetQueueTime

#Retrieves the current queue (or flush) time. Messages that have time stamps earlier than this time have already been queued to the port and cannot be invalidated.

object.**GetQueueTime() As Long**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

IDH_DirectMusicPerformance8.GetPrepareTime_dxaudio_vb
IDH_DirectMusicPerformance8.GetQueueTime_dxaudio_vb

Return Values

Returns the queue time, in clock time units.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

Remarks

Queue time is equal to the value returned by **DirectMusicPerformance8.GetLatencyTime** plus the value returned by **DirectMusicPerformance8.GetBumperLength**.

When a segment is stopped immediately, all messages that have been sent but not queued to the port buffer are flushed. If you want to resume playing the segment again at the last point heard, set the new start point to the offset of queue time within the segment when the segment was stopped.

See Also

DirectMusicPerformance8.Invalidate

DirectMusicPerformance8.GetResolvedTime

[#]Adjusts a given time to a given boundary.

```
object.GetResolvedTime( _
  ctTime As Long, _
  flags As Long _
) As Long
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

ctTime
Time to resolve, in clock time units. If this is less than the current time, the current time is used.

flags
One or more of the following **CONST_DMUS_SEGF_FLAGS** describing the resolution desired:

IDH_DirectMusicPerformance8.GetResolvedTime_dxaudio_vb

DMUS_SEGF_AFTERLATENCYTIME
Resolve to a time after the latency time.

DMUS_SEGF_AFTERPREPARETIME
Resolve to a time after the prepare time.

DMUS_SEGF_AFTERQUEUETIME
Resolve to a time after the queue time.

DMUS_SEGF_BEAT
Resolve to a time on a beat boundary.

DMUS_SEGF_GRID
Resolve to a time on a grid boundary.

DMUS_SEGF_MARKER
Resolve to a marker.

DMUS_SEGF_MEASURE
Resolve to a time on a measure boundary.

DMUS_SEGF_SEGMENTEND
Resolve to the end of the segment.

Return Values

Returns the resolved time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The time returned is the soonest time after the supplied time that falls on the specified boundary.

DirectMusicPerformance8.GetSegmentState

#Retrieves an object representing the play state of the primary segment.

```
object.GetSegmentState( _
  mtTime As Long _
) As DirectMusicSegmentState8
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

IDH_DirectMusicPerformance8.GetSegmentState_dxaudio_vb

mtTime
Time for which the segment state is to be retrieved, in music time.

Return Values

Returns a **DirectMusicSegmentState8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NOT_FOUND.

Remarks

To get the currently playing segment state, pass the time retrieved by the **DirectMusicPerformance8.GetMusicTime** method. Currently playing in this context means that it is being called into to perform messages. Because of latency, the currently playing segment state is not necessarily the one actually being heard.

Also because of latency, it is a good idea to add 150 to *mtTime* if retrieving a segment state immediately after calling **DirectMusicPerformance8.PlaySegmentEx**.

DirectMusicPerformance8.GetStyle

#Retrieves the style underlying the control segment at a given time.

```
object.GetStyle( _
  mtTime As Long, _
  mtUntil As Long _
) As DirectMusicStyle8
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

mtTime
Time for which the style is to be retrieved, in music time.

mtUntil
Variable of type **Long** that receives the music time (relative to *mtTime*) until which the style is valid. If this returns a value of 0, either the style is always valid, or it is unknown when it might become invalid. See Remarks.

Return Values

Returns a **DirectMusicStyle8** object.

IDH_DirectMusicPerformance8.GetStyle_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

Remarks

Normally the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. The object returned by the method can become invalid before the time returned in *mtUntil* if another control segment is cued.

DirectMusicPerformance8.GetTempo

[#]Retrieves the tempo at a given time.

```
object.GetTempo( _
  mtTime As Long, _
  mtUntil As Long _
) As Double
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

mtTime
Time for which to retrieve the tempo, in music time. The last tempo change before or at this time is used to determine the tempo.

mtUntil
Variable of type **Long** that receives the music time (relative to *mtTime*) until which the tempo is valid. If this returns a value of 0, either the tempo is always valid, or it is unknown when it might become invalid.

Return Values

Returns the tempo, in beats per minute. This value is in the range from DMUS_TEMPO_MIN through DMUS_TEMPO_MAX (see **CONST_DMUS**).

IDH_DirectMusicPerformance8.GetTempo_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

DirectMusicPerformance8.GetTimeSig

#Retrieves the time signature at a given time.

object.**GetTimeSig(** _
mtTime **As Long,** _
mtUntil **As Long,** _
timeSig **As DMUS_TIMESIGNATURE)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

mtTime
Value that specifies the time for which to retrieve the time signature, in music time. The last time signature change before or at this time is used to determine the time signature.

mtUntil
Variable of type **Long** that receives the music time (relative to *mtTime*) until which the time signature is valid. If this returns a value of 0, either the time signature is always valid, or it is unknown when it might become invalid. See Remarks.

timeSig
DMUS_TIMESIGNATURE type that receives information about the time signature. The **mtTime** member receives the offset of the last time signature change from the requested time, and is always 0 or less.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK

IDH_DirectMusicPerformance8.GetTimeSig_dxaudio_vb

DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

DirectMusicPerformance8.InitAudio

[#]Initializes the performance and optionally sets up a default audiopath. This method must be called before the performance can play using audiopaths.

object.**InitAudio(** _
hwnd **As Long,** _
lFlags **As CONST_DMUS_AUDIO,** _
AudioParams **As DMUS_AUDIOPARAMS,** _
[*DirectSound* **As DirectSound8],** _
[*lDefaultPathType* **As CONST_DMUSIC_STANDARD_AUDIO_PATH],** _
[*lPChannelCount* **As Long])**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

hwnd
Window handle to use for the creation of DirectSound. If 0, the foreground window is used. See Remarks.

This parameter is ignored if a **DirectSound8** object is passed to the method in *DirectSound*. In that case the application is responsible for setting the window handle by using **DirectSound8.SetCooperativeLevel**.

lFlags
Flags from the **CONST_DMUS_AUDIO** enumeration that specify requested features. If the **lValidData** member of the *AudioParams* type is not zero, this value is ignored and the requested features are specified in the **lFeatures** member of *AudioParams*.

AudioParams
DMUS_AUDIOPARAMS type that specifies parameters for the synthesizer and receives information about what parameters were set. If the **lValidData** member is zero, default parameters are set.

DirectSound
Optional **DirectSound8** object to use by default for wave output.

If this parameter is a **DirectSound8** variable set to Nothing, a DirectSound object is created and the variable is set to that object.

If this parameter is omitted, a private **DirectSound8** object is created.

See Remarks.

IDH_DirectMusicPerformance8.InitAudio_dxaudio_vb

lDefaultPathType
Value that specifies the default audiopath type. Can be zero or omitted if no default path type is wanted, or one of the constants of the **CONST_DMUSIC_STANDARD_AUDIO_PATH** enumeration.

lPChannelCount
Value that specifies the number of performance channels to allocate to the path, if *lDefaultPathType* is not zero or omitted.

Return Values

If the method fails, an error is raised. Possible values for **Err.Number** include the following:

DMUS_E_ALREADY_INITED
DMUS_E_OUTOFMEMORY

Remarks

This method can be called only once.

The performance must be terminated by using the **DirectMusicPerformance8.CloseDown** method before being released.

There should be only one **DirectSound8** object per process. If your application uses **DirectSound8** separately, it should pass that object in *DirectSound*. It must first set the cooperative level by passing DSSCL_PRIORITY to **DirectSound8.SetCooperativeLevel**.

You can pass 0 in the *hwnd* parameter to give the current foreground window handle to DirectSound. However, do not assume that the application window will be in the foreground during initialization. It is best to pass the top-level application window handle.

Parameters set in the *lFlags* and *AudioParams* parameters apply to the default audiopath and any audiopaths created subsequently.

DirectMusicPerformance8.Invalidate

[#]Flushes all queued messages whose time stamps are later than the supplied time and causes all tracks of all segments to resend their data from the given time forward.

```
object.Invalidate( _
  mtTime As Long, _
  flags As Long)
```

IDH_DirectMusicPerformance8.Invalidate_dxaudio_vb

Parts

object
Resolves to a **DirectMusicPerformance8** object.

mtTime
Time from which to invalidate, adjusted by *flags*. Setting this value to 0 causes immediate invalidation.

flags
Value that causes *mtTime* to align to measures, beats, or grids. This value can be 0 or one of the following members of **CONST_DMUS_SEGF_FLAGS**:

DMUS_SEGF_MEASURE
DMUS_SEGF_BEAT
DMUS_SEGF_GRID

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

Remarks

If *mtTime* is so long ago that it is impossible to invalidate that time, the earliest possible time is used.

See Also

DirectMusicPerformance8.GetQueueTime

DirectMusicPerformance8.IsPlaying

[#]Ascertains whether a particular segment or segment state is currently playing at the speakers.

```
object.IsPlaying( _
  segment As DirectMusicSegment8, _
  SegmentState8 As DirectMusicSegmentState8 _
) As Boolean
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

segment
DirectMusicSegment8 to check. If Nothing, check only *SegmentState8*.

IDH_DirectMusicPerformance8.IsPlaying_dxaudio_vb

SegmentState8
DirectMusicSegmentState8 to check. If Nothing, check only *segment*.

Return Values

If the method succeeds and the requested segment or segment state is playing, the return value is True. If neither is playing, or only one was requested and it is not playing, the return value is False.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK

Remarks

The method returns True only if the segment or segment state is actually playing at the speakers. Because of latency, this method might return False even though **DirectMusicPerformance8.PlaySegmentEx** has just been called on the segment. Similarly, the method returns True as long as the segment is being heard, even though all messages might already have been dispatched.

DirectMusicPerformance8.MusicToClockTime

[#]Converts time in music time format to time in clock time format.

object.**MusicToClockTime(***mtTime* **As Long) As Long**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

mtTime
Time in music time format to convert.

Return Values

Returns the time in clock time units.

IDH_DirectMusicPerformance8.MusicToClockTime_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

Remarks

This method converts a time offset from the start of the performance, not a duration. The ratio between music time and clock time units depends on the tempo, so in making the conversion, DirectMusic takes into account all tempo changes since the start of the performance. If a master tempo has been set for the performance, it is taken into account as well.

Because clock time has a greater precision than music time, a time that has been converted from clock time to music time, and then back again, will probably not have its original value.

See Also

DirectMusicPerformance8.ClockToMusicTime, Clock Time vs. Music Time

DirectMusicPerformance8.PlaySegmentEx

[#]Begins playback of a segment.

```
object.PlaySegmentEx( _
  Source As Unknown, _
  lFlags As CONST_DMUS_SEGF_FLAGS, _
  StartTime As Long, _
  [From As Unknown], _
  [AudioPath As Unknown], _
) As DirectMusicSegmentState8
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

Source
DirectMusicSegment8 object to play.

lFlags
Flags that modify the method's behavior. Can be 0 or one or more constants from the **CONST_DMUS_SEGF_FLAGS** enumeration.

StartTime

IDH_DirectMusicPerformance8.PlaySegmentEx_dxaudio_vb

Performance time at which to begin playing the segment, adjusted to any resolution boundary specified in *lFlags*. The time is in music time unless the DMUS_SEGF_REFTIME flag is set. A value of 0 causes the segment to start playing as soon as possible.

From

Optional **DirectMusicSegmentState8** or **DirectMusicAudioPath8** object to stop playing when the new segment begins. If it is an audiopath, all segment states playing on that audiopath are stopped.

AudioPath

Optional **DirectMusicAudioPath8** object that represents the audiopath on which to play the segment. If Nothing or omitted, the segment plays on the default path.

Return Values

Returns a **DirectMusicSegmentState8** object that represents the playing instance of the segment.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include the following:

DMUS_E_AUDIOPATH_INACTIVE
DMUS_E_AUDIOPATH_NOPORT
DMUS_E_NO_MASTER_CLOCK
DMUS_E_SEGMENT_INIT_FAILED
DMUS_E_TIME_PAST
DMUS_E_OUTOFMEMORY

Remarks

Segments should be greater than 250 milliseconds in length.

The boundary resolutions in *lFlags* are relative to the primary segment.

If a primary segment is scheduled to play while another primary segment is playing, the first one stops unless you set the DMUS_SEGF_QUEUE flag for the second segment, in which case it plays as soon as the first one finishes.

For more information on the exact start time of segments, see Segment Timing. For information on how the start time of segments can be affected by tempo changes, see Clock Time vs. Music Time.

If DMUS_SEGF_AUTOTRANSITION is specified in *lFlags* and a segment is playing at *StartTime* and is being interrupted, the method composes a transition between the two segments and plays it before playing *Source*.

If *Source* is a segment, a transitional segment based on a template provided at *Transition* is composed and played.

See Also

Using Segments, Using Transitions

DirectMusicPerformance8.RemoveNotificationType

#Removes a previously added notification type from the performance so that notification messages of that type are no longer sent.

object.**RemoveNotificationType(** _
type **As CONST_DMUS_NOTIFICATION_TYPE)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

type
Type of event for which notification messages are no longer to be sent. For possible values, see **CONST_DMUS_NOTIFICATION_TYPE**.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_INVALIDARG.

See Also

DirectMusicPerformance8.AddNotificationType

DirectMusicPerformance8.Reset

#Resets the port.

object.**Reset(***resetflags* **As Long)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

IDH_DirectMusicPerformance8.RemoveNotificationType_dxaudio_vb
IDH_DirectMusicPerformance8.Reset_dxaudio_vb

resetflags
Flags. If this value is 0, the method performs a GMReset. If this value is 1, the port is reset by being closed and reopened.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicPerformance8.SendCurvePMSG

[#]Sends a performance message containing information about a curve.

object.**SendCurvePMSG(** _
lTime **As Long,** _
flags **As Long,** _
channel **As Long,** _
msg **As DMUS_CURVE_PMSG)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lTime
Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags
Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

channel
Performance channel that is the destination for the message. To send the message to more than one channel, use one of the members of the **CONST_DMUS_PCHANNEL** enumeration.

msg
DMUS_CURVE_PMSG type containing information about the curve.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
DMUS_E_INVALIDARG

IDH_DirectMusicPerformance8.SendCurvePMSG_dxaudio_vb

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, the message is cued to go out immediately.

See Also

Curves

DirectMusicPerformance8.SendMIDIPMSG

#Sends a performance message containing information about a MIDI channel message not covered by other methods.

object.**SendMIDIPMSG(** _
lTime **As Long,** _
flags **As Long,** _
channel **As Long,** _
status **As Byte,** _
byte1 **As Byte,** _
byte2 **As Byte)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lTime
Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags
Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

channel
Performance channel that is the destination for the message. To send the message to more than one channel, use one of the members of the **CONST_DMUS_PCHANNEL** enumeration.

status
Standard MIDI status byte. See Remarks.

byte1
First data byte. Ignored for MIDI messages that do not require it.

IDH_DirectMusicPerformance8.SendMIDIPMSG_dxaudio_vb

byte2
Second data byte. Ignored for MIDI messages that do not require it.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, the message is cued to go out immediately.

Because the channel is specified in another parameter, *status* does not contain the channel number in the 4 lower bits, as it would in a standard MIDI message. Thus *status* is &H80 for a note-off, &H90 for a note-on, and so on. See the MIDI specification for other status bytes.

DirectMusicPerformance8.SendNotePMSG

#Sends a performance message containing information about a note.

object.**SendNotePMSG(** _
lTime **As Long,** _
flags **As Long,** _
channel **As Long,** _
msg **As DMUS_NOTE_PMSG)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lTime
Time at which the message is to play. This value is in music time unless DMUS_PMSGF_REFTIME is in *flags*. If the value of *lTime* is 0 and *flags* contains DMUS_PMSGF_REFTIME, the message plays immediately.

flags
Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

IDH_DirectMusicPerformance8.SendNotePMSG_dxaudio_vb

channel
Performance channel that is the destination for the message. To send the message to more than one channel, use one of the members of the **CONST_DMUS_PCHANNEL** enumeration.

msg
DMUS_NOTE_PMSG type containing information about the note.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
DMUS_E_INVALIDARG

Remarks

The following code example, where *perf* is a **DirectMusicPerformance8** object, sends a message to play middle C immediately.

```
Dim note As DMUS_NOTE_PMSG

note.midiValue = 60
note.mtDuration = 500
note.flags = DMUS_NOTEF_NOTEON
note.velocity = 127
Call perf.SendNotePMSG(0, DMUS_PMSGF_REFTIME, 1, note)
```

DirectMusicPerformance8.SendPatchPMSG

[#]Sends a performance message containing information about a MIDI patch change.

```
object.SendPatchPMSG( _
  lTime As Long, _
  flags As Long, _
  channel As Long, _
  instrument As Byte, _
  byte1 As Byte, _
  byte2 As Byte)
```

[#] IDH_DirectMusicPerformance8.SendPatchPMSG_dxaudio_vb

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lTime
Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags
Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

channel
Performance channel that is the destination for the message. To send the message to more than one channel, use one of the members of the **CONST_DMUS_PCHANNEL** enumeration.

instrument
Patch number to assign to the channel.

byte1
Most significant byte of bank select.

byte2
Least significant byte of bank select.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, the message is cued to go out immediately.

DirectMusicPerformance8.SendTempoPMSG

[#]Sends a performance message containing information about a tempo change.

object.**SendTempoPMSG(** _
lTime **As Long,** _
flags **As Long,** _

IDH_DirectMusicPerformance8.SendTempoPMSG_dxaudio_vb

tempo **As Double)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lTime
Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags
Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

tempo
New tempo, in beats per minute.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, the message is cued to go out immediately.

DirectMusicPerformance8.SendTimeSigPMSG

#Sends a performance message containing information about a time signature.

object.**SendTimeSigPMSG(_**
lTime **As Long, _**
flags **As Long, _**
timeSig **As DMUS_TIMESIGNATURE)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lTime

IDH_DirectMusicPerformance8.SendTimeSigPMSG_dxaudio_vb

Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags
Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

timeSig
DMUS_TIMESIGNATURE type containing information about the time signature.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, the message is cued to go out immediately.

DirectMusicPerformance8.SendTransposePMSG

#Sends a performance message causing a transposition to begin.

object.**SendTransposePMSG(** _
lTime **As Long,** _
flags **As Long,** _
channel **As Long,** _
transpose **As Integer)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lTime
Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags
Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

IDH_DirectMusicPerformance8.SendTransposePMSG_dxaudio_vb

channel
Performance channel on which the transposition is to take place. To send the message to more than one channel, use one of the members of the **CONST_DMUS_PCHANNEL** enumeration.

transpose
Number of semitones by which to transpose notes. This can be a negative value. If the transposition of a note puts it outside the standard MIDI range from 0 through 127, it does not play.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, the message is cued to go out immediately.

DirectMusicPerformance8.SetBumperLength

[#]Sets the amount of time between the time at which messages are placed in the port buffer and the time at which they begin to be processed by the port.

object.**SetBumperLength(***lMilliSeconds* **As Long)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lMilliSeconds
Desired bumper length, in milliseconds. The default value is 50.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

IDH_DirectMusicPerformance8.SetBumperLength_dxaudio_vb

See Also

DirectMusicPerformance8.GetBumperLength,
DirectMusicPerformance8.SetPrepareTime

DirectMusicPerformance8.SetDefaultAudioPath

#Sets and activates the default audiopath for the performance.

object.**SetDefaultAudioPath(_**
AudioPath **As DirectMusicAudioPath8)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

AudioPath
DirectMusicAudioPath8 object that specifies the default audiopath.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include the following:

DMUS_E_INVALIDARG
DMUS_E_AUDIOPATHS_NOT_VALID
DMUS_E_NOT_INIT

DirectMusicPerformance8.SetMasterAutoDownload

#Turns automatic downloading of instruments on or off.

object.**SetMasterAutoDownload(***b* **As Boolean)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

b
True to turn autodownloading on, False to turn it off. The default value is False.

IDH_DirectMusicPerformance8.SetDefaultAudioPath_dxaudio_vb
IDH_DirectMusicPerformance8.SetMasterAutoDownload_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicPerformance8.GetMasterAutoDownload, **DirectMusicSegment8.SetAutoDownloadEnable**

DirectMusicPerformance8.SetMasterGrooveLevel

[#]Sets a value to be added to all groove levels in the performance.

object.**SetMasterGrooveLevel(***level* **As Integer)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

level
Value to add to the groove level, in the range from –99 through 99.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_FAIL
DMUS_E_OUTOFMEMORY

See Also

DirectMusicPerformance8.GetMasterGrooveLevel

Remarks

The adjusted groove level is always in the range from 1 through 100.

IDH_DirectMusicPerformance8.SetMasterGrooveLevel_dxaudio_vb

DirectMusicPerformance8.SetMasterTempo

#Sets a scaling factor that is applied to the tempo.

object.**SetMasterTempo(***tempo* **As Single)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

tempo
Desired master tempo, in the range from 0.01 through 100.0.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

By default, the master tempo is 1. A value of 0.5 would halve the tempo, and a value of 2.0 would double it.

See Also

DirectMusicPerformance8.GetMasterTempo

DirectMusicPerformance8.SetMasterVolume

#Adjusts the master volume of the performance.

object.**SetMasterVolume(***vol* **As Long)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

vol
Master volume adjustment, in hundredths of a decibel.

IDH_DirectMusicPerformance8.SetMasterTempo_dxaudio_vb
IDH_DirectMusicPerformance8.SetMasterVolume_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The master volume is an amplification or attenuation factor applied to the default volume of the entire performance. The range of permitted values is determined by the port. For the default software synthesizer, the allowed range is +20dB to –200dB, but the useful range is +10dB to –100dB.

See Also

DirectMusicPerformance8.GetMasterVolume

DirectMusicPerformance8.SetNotificationHandle

#Sets the event handle for notifications. When signaled, the application should call the **DirectMusicPerformance8.GetNotificationPMsg** method to retrieve the notification event.

object.**SetNotificationHandle(***hnd* **As Long)**

Parts

object
Resolves to a **DirectMusicPerformance8** object.

hnd
Event handle, or 0 to clear an existing handle.

See Also

DirectXEvent8

DirectMusicPerformance8.SetPrepareTime

#Sets the approximate interval between the time at which messages are prepared and the time at which they are processed and heard.

object.**SetPrepareTime(***lMilliSeconds* **As Long)**

IDH_DirectMusicPerformance8.SetNotificationHandle_dxaudio_vb
IDH_DirectMusicPerformance8.SetPrepareTime_dxaudio_vb

Parts

object
Resolves to a **DirectMusicPerformance8** object.

lMilliSeconds
Prepare time, in milliseconds. The default value is 1000.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicPerformance8.GetPrepareTime,
DirectMusicPerformance8.SetBumperLength, Prepare Time

DirectMusicPerformance8.StopEx

#Stops playback of an object.

```
object.StopEx( _
  ObjectToStop As Unknown, _
  lStopTime As Long, _
  lFlags As Long)
```

Parts

object
Resolves to a **DirectMusicPerformance8** object.

ObjectToStop
DirectMusicSegment8, **DirectMusicSegmentState8**, or **DirectMusicAudioPath8** object to stop.

lStopTime
Time at which to stop. If the time is in the past or 0 is passed in this parameter, the object stops playing immediately.

lFlags
Flags that indicate when the stop should occur. Boundaries are in relation to the current primary segment. If this parameter is 0, the stop occurs immediately. Can contain one of the following constants from the **CONST_DMUS_SEGF_FLAGS** enumeration, or DMUS_SEGF_REFTIME combined with one other flag.

DMUS_SEGF_BEAT
Stop on the next beat boundary at or after *lStopTime*.

DMUS_SEGF_DEFAULT

IDH_DirectMusicPerformance8.StopEx_dxaudio_vb

Stop on the default boundary.

DMUS_SEGF_GRID
Stop on the next grid boundary at or after *lStopTime*.

DMUS_SEGF_MEASURE
Stop on the next measure boundary at or after *lStopTime*.

DMUS_SEGF_REFTIME
The value in *i64StopTime* is in reference time.

DMUS_SEGF_SEGMENTEND
Stop at the end of the segment.

DMUS_SEGF_MARKER
Stop at the next marker.

Error Codes

If the method fails, an error is raised.

Remarks

Stopping a segment stops all instances that are playing.

See Also

DirectMusicPerformance8.PlaySegmentEx

DirectMusicSegment8

[#]Represents a single piece of music or a template.
Segments are usually loaded by calling **DirectMusicLoader8.LoadSegment** and **DirectMusicLoader8.LoadSegmentFromResource**. They can also be composed from musical elements by using methods of the **DirectMusicComposer8** object, or created from existing segments by using the **DirectMusicSegment8.Clone** method.

The methods of the **DirectMusicSegment8** object can be grouped as follows:

Timing and looping	**GetLength**
	GetLoopPointStart
	GetLoopPointEnd
	GetRepeats
	GetStartPoint
	SetLength
	SetLoopPoints
	SetRepeats

IDH_DirectMusicSegment8_dxaudio_vb

	SetStartPoint
Instruments	**ConnectToCollection**
	Download
	Unload
Parameters	**SetAutoDownloadEnable**
	SetStandardMidiFile
	SetTempoEnable
	SetTimeSigEnable
Miscellaneous	**Clone**
	GetAudioPathConfig
	GetChordMap
	GetName
	GetStyle

DirectMusicSegment8.Clone

[#]Creates a copy of all or part of the segment.

```
object.Clone( _
  mtStart As Long, _
  mtEnd As Long _
) As DirectMusicSegment8
```

Parts

object
Resolves to a **DirectMusicSegment8** object.

mtStart
Start of the part to clone, in music time. If less than 0 or greater than the length of the segment, 0 is used.

mtEnd
End of the part to clone, in music time. If this value is past the end of the segment, the segment is cloned to the end. A value of 0 or anything less than *mtStart* also clones to the end.

Return Values

Returns a **DirectMusicSegment8** object.

IDH_DirectMusicSegment8.Clone_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_OUTOFMEMORY.

Remarks

Properties of the original segment, including start and loop points, number of repeats, and any default audiopath, are copied to the clone.

For style-based segments, if *mtStart* is greater than 0, it should be on a measure boundary.

DirectMusicSegment8.ConnectToCollection

[#]Associates a segment with a DLS instrument collection. The collection will be downloaded when the **DirectMusicSegment8.Download** method is called.

object.**ConnectToCollection(***c* **As DirectMusicCollection8)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

c
DirectMusicCollection8 object that represents instruments to be used by the segment.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND

Remarks

By default, the General MIDI collection in the Gm.dls file is connected to the segment. This method needs to be called only if the segment is to be played with custom instruments.

IDH_DirectMusicSegment8.ConnectToCollection_dxaudio_vb

DirectMusicSegment8.Download

#Downloads band and wave data to a performance or audiopath.

object.**Download(***downloadpath* **As Unknown)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

downloadpath
DirectMusicAudioPath8 or **DirectMusicPerformance8** object to which the instruments are being downloaded.

Error Codes

If the method fails, it raises an error. Possible values for **Err.Number** include the following:

DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

Remarks

All bands and wave data in the segment are downloaded.

Always call **DirectMusicSegment8.Unload** before the segment object goes out of scope or is otherwise destroyed.

See Also

DirectMusicSegment8.ConnectToCollection, **DirectMusicSegment8.Unload**, **DirectMusicSegment8.SetAutoDownloadEnable**

DirectMusicSegment8.GetAudioPathConfig

#Retrieves an object that represents the audiopath configuration for the segment.

object.**GetAudioPathConfig() As Unknown**

Parts

object

IDH_DirectMusicSegment8.Download_dxaudio_vb
IDH_DirectMusicSegment8.GetAudioPathConfig_dxaudio_vb

Resolves to a **DirectMusicSegment8** object.

Return Values

Returns an object representing the audiopath configuration. This object can be retrieved in a variable of type **IUnknown** and passed to **DirectMusicPerformance8.CreateAudioPath**.

Error Codes

If the method fails, an error is raised.

DirectMusicSegment8.GetChordMap

[#]Retrieves a chordmap.

```
object.GetChordMap( _
  lTrack As Long, _
  mtTime As Long, _
  mtUntil As Long _
) As DirectMusicChordMap8
```

Parts

object
Resolves to a **DirectMusicSegment8** object.

lTrack
Index of the chordmap track, or &H80000000 to find the track.

mtTime
Time at which to retrieve the chordmap.

mtUntil
Variable that receives the time, relative to *mtTime*, until which the chordmap is valid.

Return Values

Returns a **DirectMusicChordMap8** object.

Error Codes

If the method fails, an error is raised.

IDH_DirectMusicSegment8.GetChordMap_dxaudio_vb

DirectMusicSegment8.GetLength

#Retrieves the length of the segment.

object.**GetLength() As Long**

Parts

object
Resolves to a **DirectMusicSegment8** object.

Return Values

Returns the length of the segment, in music time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The method always returns 1 for segments created from wave files.

See Also

DirectMusicSegment8.SetLength

DirectMusicSegment8.GetLoopPointEnd

#Retrieves the point in the segment at which a repeating section is to end.

object.**GetLoopPointEnd() As Long**

Parts

object
Resolves to a **DirectMusicSegment8** object.

Return Values

Returns the end point of the loop, in music time. If this value is 0, the entire segment loops.

IDH_DirectMusicSegment8.GetLength_dxaudio_vb
IDH_DirectMusicSegment8.GetLoopPointEnd_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The section does not repeat unless the number of repetitions has been set to 1 or more by using the **DirectMusicSegment8.SetRepeats** method. By default, the entire segment repeats.

See Also

DirectMusicSegment8.GetLoopPointStart, **DirectMusicSegment8.SetLoopPoints**, **DirectMusicSegment8.GetRepeats**

DirectMusicSegment8.GetLoopPointStart

[#]Retrieves the point in the segment at which a repeating section is to start.

object.**GetLoopPointStart() As Long**

Parts

object
Resolves to a **DirectMusicSegment8** object.

Return Values

Returns the start point of the loop, in music time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The section does not repeat unless the number of repetitions has been set to 1 or more by using the **DirectMusicSegment8.SetRepeats** method. By default, the entire segment repeats.

See Also

DirectMusicSegment8.GetLoopPointEnd, **DirectMusicSegment8.SetLoopPoints**, **DirectMusicSegment8.GetRepeats**

IDH_DirectMusicSegment8.GetLoopPointStart_dxaudio_vb

DirectMusicSegment8.GetName

#Retrieves the internal name of the segment.

object.**GetName() As String**

Parts

object
Resolves to a **DirectMusicSegment8** object.

Return Values

Returns the internal name of the segment or, if it has no internal name, the filename.

Error Codes

If the method fails, an error is raised.

DirectMusicSegment8.GetRepeats

#Retrieves the number of times that the looping portion of a segment is set to repeat.

object.**GetRepeats() As Long**

Parts

object
Resolves to a **DirectMusicSegment8** object.

Return Values

Returns the number of times that the looping portion repeats.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment8.SetRepeats, **DirectMusicSegment8.SetLoopPoints**, **DirectMusicSegment8.GetLoopPointStart**, **DirectMusicSegment8.GetLoopPointEnd**, **DirectMusicSegmentState8.GetRepeats**

IDH_DirectMusicSegment8.GetName_dxaudio_vb
IDH_DirectMusicSegment8.GetRepeats_dxaudio_vb

DirectMusicSegment8.GetStartPoint

#Retrieves the point at which the segment starts playing in response to the **DirectMusicPerformance8.PlaySegmentEx** method.

object.**GetStartPoint() As Long**

Parts

object
Resolves to a **DirectMusicSegment8** object.

Return Values

Returns the start point of the segment, in music time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment8.SetStartPoint

DirectMusicSegment8.GetStyle

#Retrieves a style.

```
object.GetStyle( _
  lTrack As Long, _
  mtTime As Long, _
  mtUntil As Long _
) As DirectMusicStyle8
```

Parts

object
Resolves to a **DirectMusicSegment8** object.

lTrack
Index of the style track, or &H80000000 to find the track.

mtTime
Time at which to retrieve the style.

mtUntil
Variable that receives the time, relative to *mtTime*, until which the style is valid.

IDH_DirectMusicSegment8.GetStartPoint_dxaudio_vb
IDH_DirectMusicSegment8.GetStyle_dxaudio_vb

Return Values

Returns a **DirectMusicStyle8** object.

Error Codes

If the method fails, an error is raised.

DirectMusicSegment8.SetAutoDownloadEnable

[#]Enables or disables automatic downloading and unloading of instruments in the collection associated with the segment.

object.**SetAutoDownloadEnable(***b* **As Boolean)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

b
True to enable autodownloading, or False to disable it.

Remarks

Automatic downloading is disabled by default. When it is enabled, instruments are automatically downloaded when the segment is played, and unloaded when it is stopped.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND

See Also

DirectMusicSegment8.Download

IDH_DirectMusicSegment8.SetAutoDownloadEnable_dxaudio_vb

DirectMusicSegment8.SetLength

#Sets the length of the segment.

object.**SetLength(***mtLength* **As Long)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

mtLength
Desired length, in music time. Must be greater than 0.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_OUT_OF_RANGE.

Remarks

In most cases, applications do not need to set the length, which is automatically set when the segment is loaded. However, this method can be used to shorten a segment.

See Also

DirectMusicSegment8.GetLength

DirectMusicSegment8.SetLoopPoints

#Sets the start and end points inside the segment to repeat the number of times set by the **DirectMusicSegment8.SetRepeats** method.

object.**SetLoopPoints(***mtStart* **As Long,** *mtEnd* **As Long)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

mtStart
Point at which to begin the loop, in music time.

mtEnd
Point at which to end the loop, in music time. A value of 0 loops the entire segment.

IDH_DirectMusicSegment8.SetLength_dxaudio_vb
IDH_DirectMusicSegment8.SetLoopPoints_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_OUT_OF_RANGE.

Remarks

When the segment is played, it plays from the segment start time until *mtEnd*, then loops to *mtStart*, plays the looped portion the number of times set by **DirectMusicSegment8.SetRepeats**, and then plays to the end.

The default values are set to loop the entire segment from beginning to end.

The method fails if *mtStart* is greater than or equal to the length of the segment, or if *mtEnd* is greater than the length of the segment. If *mtEnd* is 0, *mtStart* must be 0 as well.

This method does not affect any currently playing segment states created from this segment.

A segment that is reused might be loaded from an internal cache, in which case it will have the same loop points it had the last time this value was set. It is a good idea to reset the loop points to 0 before releasing a segment that might be played again.

See Also

DirectMusicSegment8.GetLoopPointStart, **DirectMusicSegment8.GetLoopPointEnd**, **DirectMusicSegment8.SetRepeats**

DirectMusicSegment8.SetRepeats

[#]Sets the number of times that the looping portion of the segment is to repeat.

object.**SetRepeats(***lRepeats* **As Long)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

lRepeats
Number of repetitions.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

IDH_DirectMusicSegment8.SetRepeats_dxaudio_vb

See Also

DirectMusicSegment8.GetRepeats, **DirectMusicSegment8.SetLoopPoints**

DirectMusicSegment8.SetStandardMidiFile

#Informs DirectMusic that the segment is based on a standard MIDI file, not one authored specifically for DirectMusic. Calling this method ensures that certain events are handled properly when the segment is played.

object.**SetStandardMidiFile()**

Parts

object
Resolves to a **DirectMusicSegment8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The method should be called before instruments are downloaded.

DirectMusicSegment8.SetStartPoint

#Sets the point at which the segment starts playing in response to a call to the **DirectMusicPerformance8.PlaySegmentEx** method.

object.**SetStartPoint(***mtStart* **As Long)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

mtStart
Point within the segment at which it is to start playing, in music time. Must be greater than or equal to zero and less than the length of the segment.

IDH_DirectMusicSegment8.SetStandardMidiFile_dxaudio_vb
IDH_DirectMusicSegment8.SetStartPoint_dxaudio_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_OUT_OF_RANGE.

Remarks

By default, the start point is 0, meaning that the segment starts from the beginning.

The method does not affect any currently playing segment states created from this segment.

A segment that is reused might be loaded from an internal cache, in which case it will have the same start point it had the last time this value was set. It is a good idea to reset the start point to 0 before releasing a segment that might be played again.

See Also

DirectMusicSegment8.GetStartPoint, **DirectMusicSegmentState8.GetStartPoint**, **DirectMusicSegment8.SetLoopPoints**

DirectMusicSegment8.SetTempoEnable

#Enables or disables tempo messages for the segment.

object.**SetTempoEnable(***b* **As Boolean)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

b
True to enable tempo messages, or False to disable them.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND

DirectMusicSegment8.SetTimeSigEnable

#Enables or disables time signature messages for the segment.

IDH_DirectMusicSegment8.SetTempoEnable_dxaudio_vb
IDH_DirectMusicSegment8.SetTimeSigEnable_dxaudio_vb

object.**SetTempoEnable(***b* **As Boolean)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

b
True to enable time signature messages, or False to disable them.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND

DirectMusicSegment8.Unload

[#]Unloads instruments that were downloaded to the port by the **DirectMusicSegment8.Download** method.

object.**Unload(***downloadpath* **As Unknown)**

Parts

object
Resolves to a **DirectMusicSegment8** object.

downloadpath
DirectMusicAudioPath8 or **DirectMusicPerformance8** object from which to unload instruments.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment8.SetAutoDownloadEnable

IDH_DirectMusicSegment8.Unload_dxaudio_vb

DirectMusicSegmentState8

#Represents a playing instance of a segment. When the **DirectMusicPerformance8.PlaySegmentEx** method is called, it creates a **DirectMusicSegmentState8** object that enables the application to retrieve information about that instance. The object can also be passed to methods of **DirectMusicPerformance8** to determine whether a segment instance is still playing, or to stop it.

The class has the following methods:

Information	**GetRepeats**
	GetSeek
	GetSegment
	GetStartPoint
	GetStartTime

See Also

DirectMusicPerformance8.GetSegmentState

DirectMusicSegmentState8.GetRepeats

#Returns the number of times that the looping portion of the segment is set to repeat.

object.**GetRepeats() As Long**

Parts

object
Resolves to a **DirectMusicSegmentState8** object.

Return Values

Returns the repeat count. A value of 0 indicates that the segment is to play through only once, with no portion repeated.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment8.SetRepeats

IDH_DirectMusicSegmentState8_dxaudio_vb
IDH_DirectMusicSegmentState8.GetRepeats_dxaudio_vb

DirectMusicSegmentState8.GetSeek

#Retrieves the seek pointer in the segment state. This is immediately after the last point in the segment for which messages have been generated.

object.**GetSeek() As Long**

Parts

object
Resolves to a **DirectMusicSegmentState8** object.

Return Values

Returns the seek pointer, in music time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

When a segment is stopped, messages that have been sent but not yet queued to the port buffer are flushed. Therefore, if you stop a segment and then restart it at the last seek pointer, some notes are lost.

DirectMusicSegmentState8.GetSegment

#Returns an object representing the segment that owns this segment state.

object.**GetSegment() As DirectMusicSegment8**

Parts

object
Resolves to a **DirectMusicSegmentState8** object.

Return Values

Returns a **DirectMusicSegment8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

IDH_DirectMusicSegmentState8.GetSeek_dxaudio_vb
IDH_DirectMusicSegmentState8.GetSegment_dxaudio_vb

DirectMusicSegmentState8.GetStartPoint

#Returns the offset into the segment at which play began or will begin.

object.**GetStartPoint() As Long**

Parts

object
Resolves to a **DirectMusicSegmentState8** object.

Return Values

Returns the start point for this segment state, in music time. This is not the value returned by **DirectMusicSegment8.GetStartPoint** if the start point of the segment has been changed since this segment state was created. Different instances of a playing segment can have different start points.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment8.SetStartPoint, **DirectMusicSegmentState8.GetStartTime**

DirectMusicSegmentState8.GetStartTime

#Retrieves the performance time at which the segment started or will start playing.

object.**GetStartTime() As Long**

Parts

object
Resolves to a **DirectMusicSegmentState8** object.

Return Values

Returns the start time, in music time, of this instance of the segment.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_BUFFER_EMPTY.

IDH_DirectMusicSegmentState8.GetStartPoint_dxaudio_vb
IDH_DirectMusicSegmentState8.GetStartTime_dxaudio_vb

Remarks

If the segment was started from some point other than the beginning, you can retrieve the time at which the beginning of the segment would have fallen by subtracting the time returned by **DirectMusicSegmentState8.GetStartPoint** from the value returned by this method.

See Also

DirectMusicSegment8.SetStartPoint, **DirectMusicSegment8.GetStartPoint**, **DirectMusicSegmentState8.GetStartPoint**

DirectMusicStyle8

#Represents a style, which is a collection of musical data including patterns and instruments.

The object is generally obtained by using the **DirectMusicLoader8.LoadStyle** or the **DirectMusicLoader8.LoadStyleFromResource** method. It can also be obtained from the performance by using the **DirectMusicPerformance8.GetStyle** method, provided the current control segment is based on a style.

The methods of the **DirectMusicStyle8** class can be organized in the following groups:

Bands	**GetBand**
	GetBandCount
	GetBandName
	GetDefaultBand
Motifs	**GetMotif**
	GetMotifCount
	GetMotifName
Time	**GetTempo**
	GetTimeSignature

DirectMusicStyle8.GetBand

#Retrieves a band object by name.

object.**GetBand(***name* **As String) As DirectMusicBand8**

IDH_DirectMusicStyle8_dxaudio_vb
IDH_DirectMusicStyle8.GetBand_dxaudio_vb

Parts

object
Resolves to a **DirectMusicStyle8** object.

name
Name assigned to the band by the author of the style.

Return Values

Returns a **DirectMusicBand8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_FAIL.

See Also

DirectMusicStyle8.GetBandName, **DirectMusicStyle8.GetDefaultBand**

DirectMusicStyle8.GetBandCount

#Retrieves the number of bands available in the style.

object.**GetBandCount() As Long**

Parts

object
Resolves to a **DirectMusicStyle8** object.

Return Values

Returns the number of bands.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle8.GetBand, **DirectMusicStyle8.GetBandName**

DirectMusicStyle8.GetBandName

#Gets the name of a band in the style.

IDH_DirectMusicStyle8.GetBandCount_dxaudio_vb
IDH_DirectMusicStyle8.GetBandName_dxaudio_vb

object.**GetBandName(***index* **As Long) As String**

Parts

object
Resolves to a **DirectMusicStyle8** object.

index
Index of the band in the style, in the range from 1 through **DirectMusicStyle8.GetBandCount**.

Return Values

Returns the name assigned to the band by the author of the style.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle8.GetBand

DirectMusicStyle8.GetDefaultBand

[#]Retrieves the default band for the style.

object.**GetDefaultBand() As DirectMusicBand8**

Parts

object
Resolves to a **DirectMusicStyle8** object.

Return Values

Returns a **DirectMusicBand8** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle8.GetBand

IDH_DirectMusicStyle8.GetDefaultBand_dxaudio_vb

DirectMusicStyle8.GetMotif

#Creates a segment containing the named motif.

object.**GetMotif(***name* **As String) As DirectMusicSegment8**

Parts

object
Resolves to a **DirectMusicStyle8** object.

name
Name assigned to the motif by the author of the style.

Return Values

Returns a **DirectMusicSegment8** object representing the motif.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle8.GetMotifCount, **DirectMusicStyle8.GetMotifName**

DirectMusicStyle8.GetMotifCount

#Gets the number of motifs available in the style.

object.**GetMotifCount() As Long**

Parts

object
Resolves to a **DirectMusicStyle8** object.

Return Values

Returns the number of motifs in the style.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

IDH_DirectMusicStyle8.GetMotif_dxaudio_vb
IDH_DirectMusicStyle8.GetMotifCount_dxaudio_vb

DirectMusicStyle8.GetMotifName

#Gets the name of a motif in the style.

object.**GetMotifName(***index* **As Long) As String**

Parts

object
Resolves to a **DirectMusicStyle8** object.

index
Index of the motif in the style, in the range from 1 through **DirectMusicStyle8.GetMotifCount**.

Return Values

Returns the name assigned to the motif by the author of the style.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle8.GetMotif

DirectMusicStyle8.GetTempo

#Retrieves the recommended tempo of the style.

object.**GetTempo() As Double**

Parts

object
Resolves to a **DirectMusicStyle8** object.

Return Values

Returns the recommended tempo, in beats per minute.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

IDH_DirectMusicStyle8.GetMotifName_dxaudio_vb
IDH_DirectMusicStyle8.GetTempo_dxaudio_vb

DirectMusicStyle8.GetTimeSignature

#Retrieves the style's time signature.

object.**GetTimeSignature(***pTimeSig* **As DMUS_TIMESIGNATURE)**

Parts

object
Resolves to a **DirectMusicStyle8** object.

pTimeSig
DMUS_TIMESIGNATURE type that receives information about the time signature.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusic Types

This section contains information on the following types used in DirectMusic for Visual Basic.

- **DMUS_AUDIOPARAMS**
- **DMUS_CURVE_PMSG**
- **DMUS_NOTE_PMSG**
- **DMUS_NOTIFICATION_PMSG**
- **DMUS_TIMESIGNATURE**

DMUS_AUDIOPARAMS

#Describes required resources for the default synthesizer and buffers of a performance. Passed to the **DirectMusicPerformance8.InitAudio** method to request desired features and to receive information about what requests were granted.

```
Type DMUS_AUDIOPARAMS
    clsidDefaultSynth As String
    fInitNow As Boolean
    lFeatures As Long
```

IDH_DirectMusicStyle8.GetTimeSignature_dxaudio_vb
IDH_DMUS_AUDIOPARAMS_dxaudio_vb

```
    lSampleRate As Long
    lValidData As Long
    lVoices As Long
End Type
```

Members

clsidDefaultSynth
Class identifier of the default synthesizer. This is the synthesizer used by standard audiopaths and audiopaths created from configurations that request the default synthesizer. If this value is not supplied, the Microsoft software synthesizer is the default synthesizer.

fInitNow
Boolean value that specifies whether the sink and synthesizer are created immediately. If so, results are returned in this structure.

lFeatures
Flags that specify capabilities of the path. Can be 0, or one or more constants of the **CONST_DMUS_AUDIO** enumeration.

lSampleRate
Sample rate of sink and synthesizer, in the range from 11,025 to 96,000 kHz. The default value is 22,050.

lValidData
Flags that specify which members of this structure contain valid data. If **fInitNow** is True when the structure is passed, this member receives flags that specify what members received data. See **CONST_DMUS_AUDIOPARAMS**.

lVoices
Number of voices. The default value is 64.

See Also

DirectMusicPerformance8.InitAudio

DMUS_CURVE_PMSG

#Contains information about a curve message.

```
Type DMUS_CURVE_PMSG
    beat As Byte
    ccData As Byte
    curveShape As Byte
    endValue As Integer
    flags As Byte
    grid As Byte
```

IDH_DMUS_CURVE_PMSG_dxaudio_vb

```
    measure As Integer
    MergeIndex As Integer
    mtDuration As Long
    mtOriginalStart As Long
    mtResetDuration As Long
    offset As Integer
    ParamType As Integer
    resetValue As Integer
    startValue As Integer
    type As Byte
End Type
```

Members

beat
Beat count (within a measure) at which this curve occurs.

ccData
Controller number, if **type** is DMUS_CURVET_CCCURVE.

curveShape
Shape of the curve. This can be one of the values from the **CONST_DMUS_CURVES** enumeration.

endValue
End value of the curve.

flags
Flags. Can be 0 or one or more constants from the **CONST_DMUS_CURVE_FLAGS** enumeration.

grid
Grid offset from a beat at which this curve occurs. In application-created messages, should be 0.

measure
Measure in which this curve occurs. In application-created messages, should be 0.

MergeIndex
Merge index. Supported for pitch bend, volume, and expression controllers. See Remarks.

mtDuration
How long the curve lasts. This value is in music time unless DMUS_PMSGF_LOCKTOREFTIME is present in the *flags* parameter of **DirectMusicPerformance8.SendCurvePMSG**, in which case it is in milliseconds and is unaffected by a change in tempo.

mtOriginalStart
Original start time, in music time. In application-created messages, must be 0.

mtResetDuration

Length of time after the end of the curve during which a reset can take place in response to an invalidation. Ignored if DMUS_CURVE_RESET is not in the **flags** member. This value is in music time unless DMUS_PMSGF_LOCKTOREFTIME is present in the *flags* parameter of **DirectMusicPerformance8.SendCurvePMSG**, in which case it is in milliseconds and is unaffected by a change in tempo.

offset
Offset from a grid at which this curve occurs, in music time.

ParamType
MIDI parameter number. See Remarks.

resetValue
Value to set after a flush or invalidation.

startValue
Start value of the curve.

type
Type of curve. This can be one of the values from the **CONST_DMUS_CURVET** enumeration.

Remarks

An RPN or NRPN curve type in **ParamType** is stored as two bytes with seven significant bits. For example, if the MSB is 0x23 and the LSB is 0x74, the value in **ParamType** is 0x2374.

Data in **startValue**, **endValue**, and **resetValue** is limited to 14 bits. For MIDI data consisting of two seven-bit bytes, the value is stored as a word with the upper two bits empty.

All curves with **MergeIndex** of 0 override each other. If the merge index is another value, the values generated by the curve are added to the values for merge index 0. For example, if an application uses curves with indexes of 0 and 3, the 0 curves always replace each other but add to the 3 curves, and the 3 curves also always replace each other and add to the 0 curves.

See Also

DirectMusicPerformance8.SendCurvePMSG

DMUS_NOTE_PMSG

#Contains data for a music note event.

```
Type DMUS_NOTE_PMSG
    beat as Byte
    durRange As Byte
```

IDH_DMUS_NOTE_PMSG_dxaudio_vb

```
    flags As Byte
    grid As Byte
    measure As Integer
    midiValue As Byte
    mtDuration As Long
    musicValue As Integer
    offset As Integer
    playModeFlags As Byte
    subChordLevel As Byte
    timeRange As Byte
    transpose As Integer
    velocity As Byte
    velRange As Byte
End Type
```

Members

beat
Beat count (within a measure) at which this note occurs.

durRange
Range to randomize duration.

flags
Should be set to DMUS_NOTEF_NOTEON. See Remarks.

grid
Grid offset from a beat at which this note occurs.

measure
Measure in which this note occurs.

midiValue
MIDI note value.

mtDuration
Duration of the note, in music time. There are 768 ticks in a quarter note.

musicValue
Description of the note. In most play modes, this is a packed array of four-bit values, as follows:

Octave, in the range from –2 through 14. The note is transposed up or down by the octave times 12.

Chord position, in the range from 0 through 15, though it should never be above 3. The first position in the chord is 0.

Scale position, in the range from 0 through 15. Typically, it is from 0 through 2, but it is possible to have a one-note chord and have everything above the chord be interpreted as a scale position.

Accidental, in the range from –8 through 7, but typically in the range from –2 through 2. This represents an offset that takes the note out of the scale.

In the fixed play modes, the music value is a MIDI note value in the range from 0 through 127.

offset
Offset from a grid at which this note occurs, in music time.

playModeFlags
Play mode determining how the music value is related to the chord and subchord. For a list of values, see **CONST_DMUS_PLAYMODE_FLAGS**.

subChordLevel
Subchord level that the note uses.

timeRange
Range by which to randomize time.

transpose
Transposition to add to **midiValue** after conversion from **musicValue**.

velocity
Note velocity.

velRange
Range by which to randomize velocity.

Remarks

Applications sending note messages must provide values only in the **flags**, **midiValue**, **mtDuration**, and **velocity** members. The other members are of interest only to tools, which are not supported in DirectX for Visual Basic.

Normally the application sets **flags** to DMUS_NOTEF_NOTEON. The application is not responsible for sending note-off messages. When the DirectMusic output tool receives a **DMUS_NOTE_PMSG** and sees that DMUS_NOTEF_NOTEON is set, it clears the flag, adds **mtDuration** to the time stamp, and requeues the message so that the note is turned off at the appropriate time.

It is possible, however, for the application to stop a note prematurely by sending the same note on the same channel with **flags** set to 0.

See Also

DirectMusicPerformance8.SendNotePMSG, Music Values and MIDI Notes

DMUS_NOTIFICATION_PMSG

#Contains information about a notification message sent by the performance.

```
Type DMUS_NOTIFICATION_PMSG
    ctTime As Long
    lField1 As Long
    lField2 As Long
```

IDH_DMUS_NOTIFICATION_PMSG_dxaudio_vb

```
    lNotificationOption As Long
    lNotificationType As Long
    mtTime As Long
    User As Unknown
End Type
```

Members

ctTime
Time stamp of the message, in clock time.

lField1
Extra data specific to the type of notification. For DMUS_NOTIFY_ON_MEASUREANDBEAT notifications, this member returns the beat number within the measure.

lField2
Extra data specific to the type of notification. Reserved for future or application-defined use.

lNotificationOption
Identifier of the notification subtype, from the **CONST_DMUS_NOTIFICATION_SUBTYPE** enumeration.

If the notification type is DMUS_NOTIFY_ON_SEGMENT, this member can contain one of the following values:

DMUS_NOTIFICATION_SEGABORT
The segment was stopped prematurely or was removed from the primary segment queue.

DMUS_NOTIFICATION_SEGALMOSTEND
The segment has reached the end minus the prepare time.

DMUS_NOTIFICATION_SEGEND
The segment has ended.

DMUS_NOTIFICATION_SEGLOOP
The segment has looped.

DMUS_NOTIFICATION_SEGSTART
The segment has started.

If the notification type is DMUS_NOTIFY_ON_COMMAND, this member can contain one of the following values:

DMUS_NOTIFICATION_GROOVE
Groove change.

DMUS_NOTIFICATION_EMBELLISHMENT
Embellishment command (intro, fill, break, or end).

If the notification type is DMUS_NOTIFY_ON_PERFORMANCE, this member can contain one of the following values:

DMUS_NOTIFICATION_MUSICALMOSTEND

The currently playing primary segment has reached the end minus the prepare time, and no more primary segments are cued to play.

DMUS_NOTIFICATION_MUSICSTARTED
Playback has started.

DMUS_NOTIFICATION_MUSICSTOPPED
Playback has stopped.

If the notification type is DMUS_NOTIFY_ON_MEASUREANDBEAT, this member contains DMUS_NOTIFICATION_MEASUREBEAT. No other subtypes are defined.

If the notification type is DMUS_NOTIFY_ON_CHORD, this member contains DMUS_NOTIFICATION_CHORD. No other subtypes are defined.

If the notification type is DMUS_NOTIFY_ON_RECOMPOSE, this member contains DMUS_NOTIFICATION_RECOMPOSE. No other subtypes are defined.

lNotificationType
Identifier of the notification type, from the **CONST_DMUS_NOTIFICATION_TYPE** enumeration.

mtTime
Time stamp of the message, in music time.

User
Object associated with the event. See Remarks.

Remarks

For most notifications, the **User** member contains a **DirectMusicSegmentState8** object. When an application is notified of a segment event, the relevant segment can be retrieved from this object by using **DirectMusicSegmentState8.GetSegment**.

For notifications of type GUID_NOTIFICATION_PERFORMANCE, the **User** member is always Nothing.

See Also

DirectMusicPerformance8.AddNotificationType, **DirectMusicPerformance8.GetNotificationPMSG**

DMUS_TIMESIGNATURE

#Contains information about a time signature. It is passed to the **DirectMusicPerformance8.GetTimeSig** and **DirectMusicStyle8.GetTimeSignature** methods, and is also used in messages sent by the **DirectMusicPerformance8.SendTimeSigPMSG** method.

IDH_DMUS_TIMESIGNATURE_dxaudio_vb

```
Type DMUS_TIMESIGNATURE
   beat As Byte
   beatsPerMeasure As Byte
   gridsPerBeat As Integer
   mtTime As Long
End Type
```

Members

beat
Bottom of time signature.

beatsPerMeasure
Top of time signature.

gridsPerBeat
Grids (subdivisions) per beat. This value determines the timing resolution for certain music events—for example, segments cued with the DMUS_SEGF_GRID flag.

mtTime
Music time at which this time signature occurs.

DirectMusic Enumerations

DirectMusic for Visual Basic uses enumerations to group constants to take advantage of the statement completion feature of the Microsoft® Visual Studio® development environment.

This section contains reference information for the following enumerations:

- **CONST_DMUS**
- **CONST_DMUS_AUDIO**
- **CONST_DMUS_AUDIOPARAMS**
- **CONST_DMUS_COMMANDT_TYPES**
- **CONST_DMUS_COMPOSEF_FLAGS**
- **CONST_DMUS_CURVE_FLAGS**
- **CONST_DMUS_CURVES**
- **CONST_DMUS_CURVET**
- **CONST_DMUS_NOTEF_FLAGS**
- **CONST_DMUS_NOTIFICATION_SUBTYPE**
- **CONST_DMUS_NOTIFICATION_TYPE**

- **CONST_DMUS_PCHANNEL**
- **CONST_DMUS_PLAYMODE_FLAGS**
- **CONST_DMUS_PMSGF_FLAGS**
- **CONST_DMUS_SEGF_FLAGS**
- **CONST_DMUS_SHAPET_TYPES**
- **CONST_DMUSERR**
- **CONST_DMUSIC_PATH**
- **CONST_DMUSIC_STANDARD_AUDIO_PATH**

CONST_DMUS

#Contains miscellaneous constants used in DirectMusic.

```
Enum CONST_DMUS
   DMUS_MAXSUBCHORD =  8
   DMUS_TEMPO_MAX = 1000 (&H3E8)
   DMUS_TEMPO_MIN =    1
End Enum
```

Constants

DMUS_MAXSUBCHORD
Maximum number of subchords allowed in a chord.

DMUS_TEMPO_MAX
Maximum tempo, in beats per minute.

DMUS_TEMPO_MIN
Minimum tempo, in beats per minute.

CONST_DMUS_AUDIOPARAMS

#Used to specify valid members of the **DMUS_AUDIOPARAMS** type.

```
Enum CONST_DMUS_AUDIOPARAMS
   DMUS_AUDIOPARAMS_DEFAULTSYNTH =  8
   DMUS_AUDIOPARAMS_FEATURES     =  1
   DMUS_AUDIOPARAMS_SAMPLERATE   =  4
   DMUS_AUDIOPARAMS_VOICES       =  2
End Enum
```

DMUS_AUDIOPARAMS_DEFAULTSYNTH
The **clsidDefaultSynth** member contains valid data.

IDH_CONST_DMUS_dxaudio_vb
IDH_CONST_DMUS_AUDIOPARAMS_dxaudio_vb

DMUS_AUDIOPARAMS_FEATURES
The **lFeatures** member contains valid data.

DMUS_AUDIOPARAMS_SAMPLERATE
The **lSampleRate** member contains valid data.

DMUS_AUDIOPARAMS_VOICES
The **lVoices** member contains valid data.

CONST_DMUS_AUDIO

[#]Used to specify features in an audiopath.

```
Enum CONST_DMUS_AUDIO
   DMUS_AUDIOF_3D       = 1
   DMUS_AUDIOF_ALL      = 31 (&H1F)
   DMUS_AUDIOF_BUFFERS  = 32 (&h20)
   DMUS_AUDIOF_DMOS     = 8
   DMUS_AUDIOF_EAX      = 4
   DMUS_AUDIOF_ENVIRON  = 2
   DMUS_AUDIOF_STREAMING = 16 (&H10)
End Enum
```

Constants

DMUS_AUDIOF_3D
3-D buffers.

DMUS_AUDIOF_ALL
Combination of all other flags.

DMUS_AUDIOF_BUFFERS
Multiple buffers. This flag should always be set to provide full support for audiopaths.

DMUS_AUDIOF_DMOS
Additional DMOs.

DMUS_AUDIOF_EAX
EAX effects.

DMUS_AUDIOF_ENVIRON
Environmental modeling.

DMUS_AUDIOF_STREAMING
Support for streaming waves.

IDH_CONST_DMUS_AUDIO_dxaudio_vb

CONST_DMUS_COMMANDT_TYPES

#Used to identify commands that establish musical patterns.

```
Enum CONST_DMUS_COMMANDT_TYPES
    DMUS_COMMANDT_BREAK       = 3
    DMUS_COMMANDT_END         = 4
    DMUS_COMMANDT_ENDANDINTRO = 5
    DMUS_COMMANDT_FILL        = 1
    DMUS_COMMANDT_GROOVE      = 0
    DMUS_COMMANDT_INTRO       = 2
End Enum
```

Constants

DMUS_COMMANDT_BREAK
Break.

DMUS_COMMANDT_END
Ending.

DMUS_COMMANDT_ENDANDINTRO
Ending and intro.

DMUS_COMMANDT_FILL
Fill.

DMUS_COMMANDT_GROOVE
Normal groove pattern.

DMUS_COMMANDT_INTRO
Intro.

See Also

DirectMusicComposer8.AutoTransition,
DirectMusicComposer8.ComposeTransition,
DirectMusicPerformance8.GetCommand

CONST_DMUS_COMPOSEF_FLAGS

#Used as flags in the composition of transitions.

```
Enum CONST_DMUS_COMPOSEF_FLAGS
```

IDH_CONST_DMUS_COMMANDT_TYPES_dxaudio_vb
IDH_CONST_DMUS_COMPOSEF_FLAGS_dxaudio_vb

```
    DMUS_COMPOSEF_1BAR_ADDITION      =  131072 (&H20000)
    DMUS_COMPOSEF_1BAR_TRANSITION    =   32768 (&H8000)
    DMUS_COMPOSEF_AFTERPREPARETIME   =      64 (&H40)
    DMUS_COMPOSEF_ALIGN              =       1
    DMUS_COMPOSEF_BEAT               =      16 (&H10)
    DMUS_COMPOSEF_DEFAULT            =  524288 (&H80000)
    DMUS_COMPOSEF_ENTIRE_ADDITION    =   65536 (&H10000)
    DMUS_COMPOSEF_ENTIRE_TRANSITION  =   16384 (&H4000)
    DMUS_COMPOSEF_GRID               =       8
    DMUS_COMPOSEF_IMMEDIATE          =       4
    DMUS_COMPOSEF_LONG               =    8192 (&H2000)
    DMUS_COMPOSEF_MARKER             =    2048 (&H800)
    DMUS_COMPOSEF_MEASURE            =      32 (&H20)
    DMUS_COMPOSEF_MODULATE           =    4096 (&H1000)
    DMUS_COMPOSEF_NOINVALIDATE       = 1048576 (&H100000)
    DMUS_COMPOSEF_NONE               =       0
    DMUS_COMPOSEF_OVERLAP            =       2
    DMUS_COMPOSEF_SEGMENTEND         =    1024 (&H400)
    DMUS_COMPOSEF_USE_AUDIOPATH      = 2097152 (&H200000)
    DMUS_COMPOSEF_VALID_START_BEAT   =     128 (&H80)
    DMUS_COMPOSEF_VALID_START_GRID   =     256 (&H100)
    DMUS_COMPOSEF_VALID_START_MEASURE =  262144 (&H40000)
    DMUS_COMPOSEF_VALID_START_TICK   =     512 (&H200)
End Enum
```

Constants

DMUS_COMPOSEF_1BAR_ADDITION
: Include one bar of the additional transition pattern. This is the default behavior when DMUS_COMPOSEF_LONG is specified.

DMUS_COMPOSEF_1BAR_TRANSITION
: Include one bar of the transition pattern.

DMUS_COMPOSEF_AFTERPREPARETIME
: **AutoTransition** only. Use the DMUS_SEGF_AFTERPREPARETIME flag (see **CONST_DMUS_SEGF_FLAGS**) when cueing the transition.

DMUS_COMPOSEF_ALIGN
: Align transition to the time signature of the currently playing segment.

DMUS_COMPOSEF_BEAT
: **AutoTransition** only. Start transition on a beat boundary.

DMUS_COMPOSEF_DEFAULT
: Use the segment's default boundary.

DMUS_COMPOSEF_ENTIRE_ADDITION
: Include the additional transition pattern in its entirety. Used in combination with DMUS_COMPOSEF_LONG.

DMUS_COMPOSEF_ENTIRE_TRANSITION
Include the entire transition pattern.

DMUS_COMPOSEF_GRID
AutoTransition only. Start transition on a grid boundary.

DMUS_COMPOSEF_IMMEDIATE
AutoTransition only. Start transition on a music or clock time boundary.

DMUS_COMPOSEF_LONG
Composes a long transition. If this flag is not set, the length of the transition is at most one measure unless the *lCommand* parameter of **ComposeTransition** or **AutoTransition** specifies an ending and the style contains an ending of greater than one measure. If this flag is set, the length of the transition increases by one measure.

DMUS_COMPOSEF_MARKER
Play the transition at the next marker in the current segment.

DMUS_COMPOSEF_MEASURE
AutoTransition only. Start transition on a measure boundary.

DMUS_COMPOSEF_MODULATE
Compose a transition that modulates smoothly from *fromSeg* to *toSeg*, using the chord of *toSeg*.

DMUS_COMPOSEF_NOINVALIDATE
Do not invalidate segments that are playing.

DMUS_COMPOSEF_NONE
No flags. By default, the transition starts on a measure boundary.

DMUS_COMPOSEF_OVERLAP
Overlap the transition into *toSeg*. Not currently implemented.

DMUS_COMPOSEF_SEGMENTEND
Play the transition at the end of the current segment.

DMUS_COMPOSEF_USE_AUDIOPATH
Use audiopath embedded in the segments.

DMUS_COMPOSEF_VALID_START_BEAT
Allow the switch to occur on any beat. Used in conjunction with DMUS_COMPOSEF_ALIGN.

DMUS_COMPOSEF_VALID_START_GRID
Allow the switch to occur on any grid. Used in conjunction with DMUS_COMPOSEF_ALIGN.

DMUS_COMPOSEF_VALID_START_MEASURE
Allow the switch to occur on any bar. Used in combination with DMUS_COMPOSEF_ALIGN.

DMUS_COMPOSEF_VALID_START_TICK
Allow the switch to occur at any time. Used in conjunction with DMUS_COMPOSEF_ALIGN.

See Also

DirectMusicComposer8.AutoTransition,
DirectMusicComposer8.ComposeTransition

CONST_DMUS_CURVE_FLAGS

#Used in the **flags** member of the **DMUS_CURVE_PMSG** type.

```
Enum CONST_DMUS_CURVE_FLAGS
   DMUS_CURVE_RESET           = 1
   DMUS_CURVE_START_FROM_CURRENT = 2
End Enum
```

Constants

DMUS_CURVE_RESET
: The value in the **resetValue** member is to be set when the time is reached or an invalidation occurs because of a transition. If this flag is not set, the curve stays permanently at the new value.

DMUS_CURVE_START_FROM_CURRENT
: Ignore **nStartValue** and start the curve at the current value. Implemented for volume, expression, pitch bend, filter cuttoff, pan, and mod wheel.

CONST_DMUS_CURVES

#Used to define the shape of a curve in the **DMUS_CURVE_PMSG** type.

```
Enum CONST_DMUS_CURVES
   DMUS_CURVES_EXP      = 2
   DMUS_CURVES_INSTANT  = 1
   DMUS_CURVES_LINEAR   = 0
   DMUS_CURVES_LOG      = 3
   DMUS_CURVES_SINE     = 4
End Enum
```

Constants

DMUS_CURVES_EXP
: Exponential curve shape.

DMUS_CURVES_INSTANT
: Instant curve shape (beginning and end of curve happen at essentially the same time).

IDH_CONST_DMUS_CURVE_FLAGS_dxaudio_vb
IDH_CONST_DMUS_CURVES_dxaudio_vb

DMUS_CURVES_LINEAR
Linear curve shape.

DMUS_CURVES_LOG
Logarithmic curve shape.

DMUS_CURVES_SINE
Sine curve shape.

CONST_DMUS_CURVET

#Used to describe the MIDI controller in the **DMUS_CURVE_PMSG** type.

```
Enum CONST_DMUS_CURVET
    DMUS_CURVET_CCCURVE   = 4
    DMUS_CURVET_MATCURVE  = 5
    DMUS_CURVET_NRPNCURVE = 8
    DMUS_CURVET_PATCURVE  = 6
    DMUS_CURVET_PBCURVE   = 3
    DMUS_CURVET_RPNCURVE  = 7
End Enum
```

Constants

DMUS_CURVET_CCCURVE
Continuous controller curve (MIDI Control Change channel voice message; status byte &HB*n*, where *n* is the channel number).

DMUS_CURVET_MATCURVE
Monophonic aftertouch curve (MIDI Channel Pressure channel voice message; status byte &HD*n*).

DMUS_CURVET_NRPNCURVE
NRPN curve of type defined in the **ParamType** member.

DMUS_CURVET_PATCURVE
Polyphonic aftertouch curve (MIDI Poly Key Pressure channel voice message, status byte &HD*n*).

DMUS_CURVET_PBCURVE
Pitch-bend curve (MIDI Pitch Bend channel voice message; status byte &HE*n*).

DMUS_CURVET_RPNCURVE
RPN curve of type defined in **ParamType**.

CONST_DMUS_NOTEF_FLAGS

#Used in note messages.

IDH_CONST_DMUS_CURVET_dxaudio_vb
IDH_CONST_DMUS_NOTEF_FLAGS_dxaudio_vb

```
Enum CONST_DMUS_NOTEF_FLAGS
   DMUS_NOTEF_NOTEON = 1
End Enum
```

Constants

DMUS_NOTEF_NOTEON
See the Remarks for **DMUS_NOTE_PMSG**.

CONST_DMUS_NOTIFICATION_SUBTYPE

#Provide information about the musical events reported in notification messages.

```
Enum CONST_DMUS_NOTIFICATION_SUBTYPE
   DMUS_NOTIFICATION_CHORD         = 0
   DMUS_NOTIFICATION_EMBELLISHMENT  = 1
   DMUS_NOTIFICATION_GROOVE        = 0
   DMUS_NOTIFICATION_MEASUREBEAT    = 0
   DMUS_NOTIFICATION_MUSICALMOSTEND = 2
   DMUS_NOTIFICATION_MUSICSTARTED   = 0
   DMUS_NOTIFICATION_MUSICSTOPPED   = 1
   DMUS_NOTIFICATION_RECOMPOSE      = 0
   DMUS_NOTIFICATION_SEGABORT       = 4
   DMUS_NOTIFICATION_SEGALMOSTEND   = 2
   DMUS_NOTIFICATION_SEGEND         = 1
   DMUS_NOTIFICATION_SEGLOOP        = 3
   DMUS_NOTIFICATION_SEGSTART       = 0
End Enum
```

For an explanation of the constants, see **DMUS_NOTIFICATION_PMSG**.

CONST_DMUS_NOTIFICATION_TYPE

#Used to identify a notification type. These constants are passed to the **DirectMusicPerformance8.AddNotificationType** and **DirectMusicPerformance8.RemoveNotificationType** methods and identify the notification type in messages retrieved by **DirectMusicPerformance8.GetNotificationPMSG**.

IDH_CONST_DMUS_NOTIFICATION_SUBTYPE_dxaudio_vb
IDH_CONST_DMUS_NOTIFICATION_TYPE_dxaudio_vb

```
Enum CONST_DMUS_NOTIFICATION_TYPE
   DMUS_NOTIFY_ON_CHORD         = 1
   DMUS_NOTIFY_ON_COMMAND       = 2
   DMUS_NOTIFY_ON_MEASUREANDBEAT = 3
   DMUS_NOTIFY_ON_PERFORMANCE    = 4
   DMUS_NOTIFY_ON_RECOMPOSE =      0
   DMUS_NOTIFY_ON_SEGMENT       = 5
End Enum
```

Constants

DMUS_NOTIFY_ON_CHORD
Chord change.

DMUS_NOTIFY_ON_COMMAND
Command event.

DMUS_NOTIFY_ON_MEASUREANDBEAT
Measure and beat event.

DMUS_NOTIFY_ON_PERFORMANCE
Performance event. When this value is found in the **lNotificationType** member of a **DMUS_NOTIFICATION_PMSG** type, the event is further defined in the **lNotificationOption** member.

DMUS_NOTIFY_ON_RECOMPOSE
Recomposition event. When this value is found in the **lNotificationType** of a **DMUS_NOTIFICATION_PMSG** type, the event is further defined in the **lNotificationOption** member.

DMUS_NOTIFY_ON_SEGMENT
Segment event. When this value is found in the **lNotificationType** of a **DMUS_NOTIFICATION_PMSG** type, the event is further defined in the **lNotificationOption** member.

See Also

DMUS_NOTIFICATION_PMSG

CONST_DMUS_PCHANNEL

[#]Used to direct messages to multiple performance channels.
DMUS_PCHANNEL_ALL is used to search multiple channels for objects in the audiopath.

```
Enum CONST_DMUS_PCHANNEL
   DMUS_PCHANNEL_ALL               = -5 (&HFFFFFFFB)
   DMUS_PCHANNEL_BROADCAST_AUDIOPATH   = -2 (&HFFFFFFFE)
```

IDH_CONST_DMUS_PCHANNEL_dxaudio_vb

```
    DMUS_PCHANNEL_BROADCAST_GROUPS      = -4 (&HFFFFFFFC)
    DMUS_PCHANNEL_BROADCAST_PERFORMANCE = -1 (&HFFFFFFFF)
    DMUS_PCHANNEL_BROADCAST_SEGMENT     = -3 (&HFFFFFFFD)
  End Enum
```

Constants

DMUS_PCHANNEL_ALL
: Search all channels.

DMUS_PCHANNEL_BROADCAST_AUDIOPATH
: Send a copy of the message to all channels of the audiopath.

DMUS_PCHANNEL_BROADCAST_GROUPS
: Send a copy of the message to each channel group in the performance. Used for messages that need to be sent only once per channel group, such as system exclusive messages.

DMUS_PCHANNEL_BROADCAST_PERFORMANCE
: Send a copy of the message to all channels of the performance.

DMUS_PCHANNEL_BROADCAST_SEGMENT
: Send a copy of the message to all channels of the segment.

CONST_DMUS_PLAYMODE_FLAGS

[#]Used to set the play mode in a **DMUS_NOTE_PMSG** message type. The play mode determines how the note is transposed to the current chord before it is converted to a MIDI note.

```
  Enum CONST_DMUS_PLAYMODE_FLAGS
    DMUS_PLAYMODE_ALWAYSPLAY     = 14
    DMUS_PLAYMODE_CHORD_INTERVALS = 8
    DMUS_PLAYMODE_CHORD_ROOT     = 2
    DMUS_PLAYMODE_FIXED          = 0
    DMUS_PLAYMODE_FIXEDTOCHORD   = 2
    DMUS_PLAYMODE_FIXEDTOKEY     = 1
    DMUS_PLAYMODE_KEY_ROOT       = 1
    DMUS_PLAYMODE_MELODIC        = 6
    DMUS_PLAYMODE_NONE           = 16 (&H10)
    DMUS_PLAYMODE_NORMALCHORD    = 10
    DMUS_PLAYMODE_PEDALPOINT     = 5
    DMUS_PLAYMODE_SCALE_INTERVALS = 4
  End Enum
```

[#] IDH_CONST_DMUS_PLAYMODE_FLAGS_dxaudio_vb

Constants

The following members are the basic flags:

DMUS_PLAYMODE_CHORD_INTERVALS
: Use chord intervals from chord pattern.

DMUS_PLAYMODE_CHORD_ROOT
: Transpose over the chord root.

DMUS_PLAYMODE_KEY_ROOT
: Transpose over the key root.

DMUS_PLAYMODE_NONE
: No mode. Indicates that the parent part's mode should be used.

DMUS_PLAYMODE_SCALE_INTERVALS
: Use scale intervals from scale pattern.

The following members represent combinations of the basic flags:

DMUS_PLAYMODE_ALWAYSPLAY
: Combination of DMUS_PLAYMODE_SCALE_INTERVALS, DMUS_PLAYMODE_CHORD_INTERVALS, and DMUS_PLAYMODE_CHORD_ROOT. If it is desirable to play a note that is above the top of the chord, this mode finds a position for the note by using intervals from the scale. Essentially, this mode is a combination of the normal and melodic playback modes, in which a failure in normal mode causes a second try in melodic mode.

DMUS_PLAYMODE_FIXED
: Interpret the music value as a MIDI value. This is defined as 0 and signifies the absence of other flags. This flag is used for drums, sound effects, and sequenced notes that should not be transposed by the chord or scale.

DMUS_PLAYMODE_FIXEDTOCHORD
: Same as DMUS_PLAYMODE_CHORD_ROOT. The music value is a fixed MIDI value, but it is transposed over the chord root.

DMUS_PLAYMODE_FIXEDTOKEY
: Same as DMUS_PLAYMODE_KEY_ROOT. The music value is a fixed MIDI value, but it is transposed over the key root.

DMUS_PLAYMODE_MELODIC
: Combination of DMUS_PLAYMODE_CHORD_ROOT and DMUS_PLAYMODE_SCALE_INTERVALS. The chord root is used, but the notes track only the intervals in the scale. The key root and chord intervals are ignored. This is useful for melodic lines that play relative to the chord root.

DMUS_PLAYMODE_NORMALCHORD
: Combination of DMUS_PLAYMODE_CHORD_ROOT and DMUS_PLAYMODE_CHORD_INTERVALS. This is the prevalent playback mode. The notes track the intervals in the chord, which is based on the chord root. If there is a scale component to the music value, the additional intervals are pulled from the scale and added. If the chord does not have an interval to match the chord component of the music value, the note is silent.

DMUS_PLAYMODE_PEDALPOINT
Combination of DMUS_PLAYMODE_KEY_ROOT and DMUS_PLAYMODE_SCALE_INTERVALS. The key root is used, and the notes track only the intervals in the scale. The chord root and intervals are ignored. This is useful for melodic lines that play relative to the key root.

CONST_DMUS_PMSGF_FLAGS

#The members of the **CONST_DMUS_PMSGF_FLAGS** enumeration are used in the various message-sending methods of **DirectMusicPerformance8**.

```
Enum CONST_DMUS_PMSGF_FLAGS
   DMUS_PMSGF_DX8          = 128 (&H80)
   DMUS_PMSGF_LOCKTOREFTIME  =  64 (&H40)
   DMUS_PMSGF_MUSICTIME     =   2
   DMUS_PMSGF_REFTIME       =   1
   DMUS_PMSGF_TOOL_ATTIME    =  16 (&H10)
   DMUS_PMSGF_TOOL_FLUSH     =  32 (&H20)
   DMUS_PMSGF_TOOL_IMMEDIATE =   4
   DMUS_PMSGF_TOOL_QUEUE     =   8
End Enum
```

Constants

DMUS_PMSGF_DX8
Message has valid members not present in versions prior to DirectX 8.0. It is not necessary to set this flag in DirectX for Visual Basic.

DMUS_PMSGF_LOCKTOREFTIME
The clock time of the message cannot be altered by a tempo change.

DMUS_PMSGF_REFTIME
The time stamp is in clock time.

DMUS_PMSGF_MUSICTIME
The time stamp is in music time.

DMUS_PMSGF_TOOL_IMMEDIATE
DMUS_PMSGF_TOOL_QUEUE
DMUS_PMSGF_TOOL_ATTIME
DMUS_PMSGF_TOOL_FLUSH
See Remarks.

Remarks

Because DirectX for Visual Basic does not support DirectMusic tools, only DMUS_PMSGF_REFTIME and DMUS_PMSGF_MUSICTIME are currently valid.

IDH_CONST_DMUS_PMSGF_FLAGS_dxaudio_vb

Because the time of messages is in music time by default, DMUS_PMSGF_REFTIME is the only flag that applications normally use.

CONST_DMUS_SEGF_FLAGS

#The members of the **CONST_DMUS_SEGF_FLAGS** enumeration are used in various methods of the **DirectMusicPerformance8** object to control the timing and other aspects of actions on a segment.

```
Enum CONST_DMUS_SEGF_FLAGS
   DMUS_SEGF_AFTERLATENCYTIME    =   4194304 (&H400000)
   DMUS_SEGF_AFTERPREPARETIME    =      1024 (&H400)
   DMUS_SEGF_AFTERQUEUETIME      =   2097152 (&H200000)
   DMUS_SEGF_ALIGN               =     65536 (&H10000)
   DMUS_SEGF_AUTOTRANSITION      =   1048576 (&H100000)
   DMUS_SEGF_BEAT                =      4096 (&H1000)
   DMUS_SEGF_CONTROL             =       512 (&H200)
   DMUS_SEGF_DEFAULT             =     16384 (&H4000)
   DMUS_SEGF_GRID                =      2048 (&H800)
   DMUS_SEGF_MARKER              =  16777216 (&H1000000)
   DMUS_SEGF_MEASURE             =      8192 (&H2000)
   DMUS_SEGF_NOINVALIDATE        =     32768 (&H8000)
   DMUS_SEGF_QUEUE               =       256 (&H100)
   DMUS_SEGF_REFTIME             =        64 (&H40)
   DMUS_SEGF_SECONDARY           =       128 (&H80)
   DMUS_SEGF_SEGMENTEND          =   8388608 (&H800000)
   DMUS_SEGF_TIMESIG_ALWAYS      =  33554432 (&H2000000)
   DMUS_SEGF_USE_AUDIOPATH       =  67108864 (&H4000000)
   DMUS_SEGF_VALID_START_BEAT    =    131072 (&H20000)
   DMUS_SEGF_VALID_START_GRID    =    262144 (&H40000)
   DMUS_SEGF_VALID_START_MEASURE = 134217728 (&H8000000
   DMUS_SEGF_VALID_START_TICK    =    524288 (&H80000)
End Enum
```

Constants

DMUS_SEGF_AFTERLATENCYTIME
: Play after the latency time. Because this is true for all segments, this flag currently has no effect.

DMUS_SEGF_AFTERPREPARETIME
: Play after the prepare time.

DMUS_SEGF_AFTERQUEUETIME
: Play after the queue time. This is the default for primary segments. Ignored if DMUS_SEGF_AFTERPREPARETIME is also set.

IDH_CONST_DMUS_SEGF_FLAGS_dxaudio_vb

DMUS_SEGF_ALIGN
The beginning of the segment can be aligned with a boundary, such as measure or beat, that has already passed. For this to happen, the segment must have a valid start point that falls before the next boundary. Start points can be defined in the segment, or one of the DMUS_SEGF_VALID_START_xxx flags can be used to define the granularity of valid start points. Any DMUS_SEGF_VALID_START_xxx flag takes effect only if a valid start point is not defined in the segment.

DMUS_SEGF_AUTOTRANSITION
Compose and play a transition segment, using the transition template.

DMUS_SEGF_BEAT
Play on a beat boundary.

DMUS_SEGF_CONTROL
Play as a control segment (secondary segments only). See Remarks.

DMUS_SEGF_DEFAULT
Use segment's default boundary.

DMUS_SEGF_GRID
Play on a grid boundary.

DMUS_SEGF_MARKER
Play at next marker in the primary segment. If there are no markers, use other resolution flags.

DMUS_SEGF_MEASURE
Play on a measure boundary.

DMUS_SEGF_NOINVALIDATE
Setting this flag in **DirectMusicPerformance8.PlaySegmentEx** for a primary or control segment causes the new segment not to cause an invalidation. Without this flag, an invalidation occurs, cutting off and resetting any currently playing curve or note. This flag should be combined with DMUS_SEGF_AFTERPREPARETIME so that notes in the new segment do not play over notes played by the old segment.

DMUS_SEGF_QUEUE
Put at the end of the primary segment queue (primary segment only).

DMUS_SEGF_REFTIME
Time parameter is in clock time.

DMUS_SEGF_SECONDARY
Secondary segment.

DMUS_SEGF_SEGMENTEND
Play at the end of the primary segment that is playing at the start time. Any segments already queued after the currently playing primary segment are flushed. If no primary segment is playing, use other resolution flags.

DMUS_SEGF_TIMESIG_ALWAYS
Align start time with current time signature, even if there is no primary segment.

DMUS_SEGF_USE_AUDIOPATH

Use the audiopath embedded in the segment. Automatic downloading of bands must be enabled to ensure that the sounds play correctly.

DMUS_SEGF_VALID_START_BEAT
Allow the start to occur on any beat. Used in combination with DMUS_SEGF_ALIGN.

DMUS_SEGF_VALID_START_GRID
Allow the start to occur on any grid. Used in combination with DMUS_SEGF_ALIGN.

DMUS_SEGF_VALID_START_MEASURE
Allow the start to occur on any measure. Used in combination with DMUS_SEGF_ALIGN.

DMUS_SEGF_VALID_START_TICK
Allow the start to occur any time. Used in combination with DMUS_SEGF_ALIGN.

Remarks

Normally the primary segment is the control segment. The DMUS_SEGF_CONTROL flag can be used to make a secondary segment the control segment. There should be only one control segment at a time. (Although it is possible to create multiple control segments, there is no guarantee of which one will actually be used by DirectMusic as the control segment.) By default, only the control segment sends tempo messages.

If the DMUS_SEGF_CONTROL flag is set, DMUS_SEGF_SECONDARY is assumed.

No more than one flag from each of the following groups should be specified.

Boundary
This flag controls the point in the currently playing primary segment at which the start point of the cued segment falls. It can be combined with DMUS_SEGF_MARKER, in which case the boundary flag will be used only if no marker exists in the primary segment.

DMUS_SEGF_BEAT
DMUS_SEGF_DEFAULT
DMUS_SEGF_GRID
DMUS_SEGF_MEASURE
DMUS_SEGF_QUEUE
DMUS_SEGF_SEGMENTEND

Alignment
This flag controls the segment start time of the cued segment, when its play time falls in the past. It must be combined with DMUS_SEGF_ALIGN.

DMUS_SEGF_VALID_START_BEAT
DMUS_SEGF_VALID_START_GRID

DMUS_SEGF_VALID_START_MEASURE
DMUS_SEGF_VALID_START_TICK

It is possible to combine one flag from each group. For example, combining DMUS_SEGF_MEASURE with DMUS_SEGF_ALIGN and DMUS_SEGF_VALID_START_BEAT causes the start point of the cued segment to fall at a measure boundary in the current primary segment. If this boundary has already passed, the cued segment starts playing at the next beat boundary within itself that is not aligned to a past time. For more information, see Segment Timing.

See Also

DirectMusicPerformance8.GetResolvedTime, **DirectMusicPerformance8.Invalidate**, **DirectMusicPerformance8.PlaySegmentEx**, **DirectMusicPerformance8.StopEx**

CONST_DMUS_SHAPET_TYPES

[#]The members of the **CONST_DMUS_SHAPET_TYPES** enumeration are used in the *wShape* parameter of the **DirectMusicComposer8.ComposeSegmentFromShape** and **DirectMusicComposer8.ComposeTemplateFromShape** methods to specify the desired pattern of the groove level.

```
Enum CONST_DMUS_SHAPET_TYPES
   DMUS_SHAPET_FALLING  = 0
   DMUS_SHAPET_LEVEL    = 1
   DMUS_SHAPET_LOOPABLE = 2
   DMUS_SHAPET_LOUD     = 3
   DMUS_SHAPET_PEAKING  = 5
   DMUS_SHAPET_QUIET    = 4
   DMUS_SHAPET_RANDOM   = 6
   DMUS_SHAPET_RISING   = 7
   DMUS_SHAPET_SONG     = 8
End Enum
```

Constants

DMUS_SHAPET_FALLING
The groove level falls.

DMUS_SHAPET_LEVEL
The groove level remains even.

DMUS_SHAPET_LOOPABLE
The segment is arranged to loop back to the beginning.

IDH_CONST_DMUS_SHAPET_TYPES_dxaudio_vb

DMUS_SHAPET_LOUD
The groove level is high.

DMUS_SHAPET_PEAKING
The groove level rises to a peak, and then falls.

DMUS_SHAPET_QUIET
The groove level is low.

DMUS_SHAPET_RANDOM
The groove level is random.

DMUS_SHAPET_RISING
The groove level rises.

DMUS_SHAPET_SONG
The segment is in a song form. Several phrases of 6 to 8 bars are composed and put together to give a verse-chorus effect, with variations in groove level.

CONST_DMUSERR

#The **CONST_DMUSERR** enumeration represents DirectMusic error codes. For a list of values, see DirectMusic Error Codes.

CONST_DMUSIC_PATH

#Used to specify the stage in the audiopath from which to retrieve an object. Some of these flags are valid when using the **DirectMusicAudioPath8.GetObjectInPath** method.

Not all objects that can be retrieved are supported in DirectX for Visual Basic. For more information, see Retrieving Objects from an Audiopath.

```
Enum CONST_DMUSIC_PATH
    DMUS_PATH_AUDIOPATH         =  8192 (&H2000)
    DMUS_PATH_AUDIOPATH_GRAPH   =  8704 (&H2200)
    DMUS_PATH_AUDIOPATH_TOOL    =  8960 (&H2300)
    DMUS_PATH_BUFFER            = 24576 (&H6000)
    DMUS_PATH_BUFFER_DMO        = 24832 (&H6100)
    DMUS_PATH_MIXIN_BUFFER      = 28672 (&H7000)
    DMUS_PATH_MIXIN_BUFFER_DMO  = 28928 (&H7100)
    DMUS_PATH_PERFORMANCE       = 12288 (&H3000)
    DMUS_PATH_PERFORMANCE_GRAPH = 12800 (&H3200)
    DMUS_PATH_PERFORMANCE_TOOL  = 13056 (&H3300)
    DMUS_PATH_PORT              = 16384 (&H4000)
    DMUS_PATH_PRIMARY_BUFFER    = 32768 (&H8000)
    DMUS_PATH_SEGMENT           =  4096 (&H1000)
    DMUS_PATH_SEGMENT_GRAPH     =  4608 (&H1200)
```

IDH_CONST_DMUSERR_dxaudio_vb
IDH_CONST_DMUSIC_PATH_dxaudio_vb

```
    DMUS_PATH_SEGMENT_TOOL      = 4864 (&H1300)
    DMUS_PATH_SEGMENT_TRACK     = 4352 (&H1100)
End Enum
```

Constants

DMUS_PATH_AUDIOPATH
Get the audiopath from a segment state.

DMUS_PATH_AUDIOPATH_GRAPH
Get the audiopath tool graph. One is created if none exists.

DMUS_PATH_AUDIOPATH_TOOL
Get a tool from the audiopath tool graph.

DMUS_PATH_BUFFER
Get a DirectSound buffer.

DMUS_PATH_BUFFER_DMO
Get a DMO in the buffer.

DMUS_PATH_MIXIN_BUFFER
Get a global mix-in buffer.

DMUS_PATH_MIXIN_BUFFER_DMO
Get a DMO in a global mix-in buffer.

DMUS_PATH_PERFORMANCE
Get the performance.

DMUS_PATH_PERFORMANCE_GRAPH
Get the performance tool graph. One is created if none exists.

DMUS_PATH_PERFORMANCE_TOOL
Get a tool in the performance graph.

DMUS_PATH_PORT
Get the synthesizer.

DMUS_PATH_PRIMARY_BUFFER
Get the primary buffer.

DMUS_PATH_SEGMENT
Get the segment that owns this segment state.

DMUS_PATH_SEGMENT_GRAPH
Get the segment tool graph. One is created if none exists.

DMUS_PATH_SEGMENT_TOOL
Get a tool from the segment graph. See Remarks.

DMUS_PATH_SEGMENT_TRACK
Get a track from the segment.

CONST_DMUSIC_STANDARD_AUDIO_PATH[#]Used to specify the audiopath to create by using **DirectMusicPerformance8.CreateStandardAudioPath** or **DirectMusicPerformance8.InitAudio**.

```
Enum CONST_DMUSIC_STANDARD_AUDIO_PATH
   DMUS_APATH_DYNAMIC_3D           = 6
   DMUS_APATH_DYNAMIC_MONO         = 7
   DMUS_APATH_DYNAMIC_STEREO       = 8
   DMUS_APATH_SHARED_STEREOPLUSREVERB = 1
End Enum
```

Constants

DMUS_APATH_DYNAMIC_3D
An audiopath with one bus from the synthesizer feeding to a 3-D buffer.

DMUS_APATH_DYNAMIC_MONO
Audio environment with one mono buffer.

DMUS_APATH_DYNAMIC_STEREO
Audio environment with one stereo buffer.

DMUS_APATH_SHARED_STEREOPLUSREVERB
Audio environment with one stereo buffer and one buffer with environmental reverb.

DirectMusic Error Codes

This section provides a brief explanation of the various error codes that can be returned by DirectMusic methods. For a list of the specific codes that each method can return, see the individual method descriptions. These lists are not necessarily comprehensive.

Error codes are presented in the following sections:

- DirectMusic Error Codes By Number
- DirectMusic Error Codes By Name

DirectMusic Error Codes By Number

The following table lists DirectMusic error codes sorted by value. For a description, click on the constant.

Hexadecimal	**Constant**
&H00001AE	DMUS_E_NOINTERFACE

[#] IDH_CONST_DMUSIC_STANDARD_AUDIO_PATH_dxaudio_vb

&H00001BD	DMUS_E_NOTIMPL
&H80070057	DMUS_E_FAIL
&H88781101	DMUS_E_DRIVER_FAILED
&H88781102	DMUS_E_PORTS_OPEN
&H88781103	DMUS_E_DEVICE_IN_USE
&H88781104	DMUS_E_INSUFFICIENTBUFFER
&H88781105	DMUS_E_BUFFERNOTSET
&H88781106	DMUS_E_BUFFERNOTAVAILABLE
&H88781108	DMUS_E_NOTADLSCOL
&H88781109	DMUS_E_INVALIDOFFSET
&H88781111	DMUS_E_ALREADY_LOADED
&H88781113	DMUS_E_INVALIDPOS
&H88781114	DMUS_E_INVALIDPATCH
&H88781115	DMUS_E_CANNOTSEEK
&H88781116	DMUS_E_CANNOTWRITE
&H88781117	DMUS_E_CHUNKNOTFOUND
&H88781119	DMUS_E_INVALID_DOWNLOADID
&H88781120	DMUS_E_NOT_DOWNLOADED_TO_PORT
&H88781121	DMUS_E_ALREADY_DOWNLOADED
&H88781122	DMUS_E_UNKNOWNDOWNLOAD
&H88781123	DMUS_E_SET_UNSUPPORTED
&H88781124	DMUS_E_GET_UNSUPPORTED
&H88781125	DMUS_E_NOTMONO
&H88781126	DMUS_E_BADARTICULATION
&H88781127	DMUS_E_BADINSTRUMENT
&H88781128	DMUS_E_BADWAVELINK
&H88781128	DMUS_E_BUFFER_EMPTY
&H8878112A	DMUS_E_NOTPCM
&H8878112B	DMUS_E_BADWAVE
&H8878112C	DMUS_E_BADOFFSETTABLE
&H8878112D	DMUS_E_UNKNOWN_PROPERTY
&H8878112E	DMUS_E_NOSYNTHSINK
&H8878112F	DMUS_E_ALREADYOPEN
&H88781130	DMUS_E_ALREADYCLOSED
&H88781131	DMUS_E_SYNTHNOTCONFIGURED
&H88781132	DMUS_E_SYNTHACTIVE
&H88781133	DMUS_E_CANNOTREAD
&H88781134	DMUS_E_DMUSIC_RELEASED

&H88781136	DMUS_E_BUFFER_FULL
&H88781137	DMUS_E_PORT_NOT_CAPTURE
&H88781138	DMUS_E_PORT_NOT_RENDER
&H88781139	DMUS_E_DSOUND_NOT_SET
&H8878113A	DMUS_E_ALREADY_ACTIVATED
&H8878113B	DMUS_E_INVALIDBUFFER
&H8878113C	DMUS_E_WAVEFORMATNOTSUPPORTED
&H8878113D	DMUS_E_SYNTHINACTIVE
&H8878113E	DMUS_E_DSOUND_ALREADY_SET
&H8878113F	DMUS_E_INVALID_EVENT
&H88781150	DMUS_E_UNSUPPORTED_STREAM
&H88781151	DMUS_E_ALREADY_INITED
&H88781152	DMUS_E_INVALID_BAND
&H88781155	DMUS_E_TRACK_HDR_NOT_FIRST_CK
&H88781156	DMUS_E_TOOL_HDR_NOT_FIRST_CK
&H88781157	DMUS_E_INVALID_TRACK_HDR
&H88781158	DMUS_E_INVALID_TOOL_HDR
&H88781159	DMUS_E_ALL_TOOLS_FAILED
&H88781160	DMUS_E_ALL_TRACKS_FAILED
&H88781161	DMUS_E_NOT_FOUND
&H88781162	DMUS_E_NOT_INIT
&H88781163	DMUS_E_TYPE_DISABLED
&H88781164	DMUS_E_TYPE_UNSUPPORTED
&H88781165	DMUS_E_TIME_PAST
&H88781166	DMUS_E_TRACK_NOT_FOUND
&H88781170	DMUS_E_NO_MASTER_CLOCK
&H88781170	DMUS_E_NOARTICULATION
&H88781180	DMUS_E_LOADER_NOCLASSID
&H88781181	DMUS_E_LOADER_BADPATH
&H88781182	DMUS_E_LOADER_FAILEDOPEN
&H88781183	DMUS_E_LOADER_FORMATNOTSUPPORTED
&H88781184	DMUS_E_LOADER_FAILEDCREATE
&H88781185	DMUS_E_LOADER_OBJECTNOTFOUND
&H88781186	DMUS_E_LOADER_NOFILENAME
&H88781200	DMUS_E_INVALIDFILE
&H88781201	DMUS_E_ALREADY_EXISTS
&H88781202	DMUS_E_OUT_OF_RANGE
&H88781203	DMUS_E_SEGMENT_INIT_FAILED

&H88781204	DMUS_E_ALREADY_SENT
&H88781205	DMUS_E_CANNOT_FREE
&H88781206	DMUS_E_CANNOT_OPEN_PORT
&H88781207	DMUS_E_CANNOT_CONVERT
&H88781210	DMUS_E_DESCEND_CHUNK_FAIL
&H88781211	DMUS_E_NOT_LOADED
&H88781220	DMUS_E_INVALID_SEGMENTTRIGGERTRACK
&H88781221	DMUS_E_INVALID_LYRICSTRACK
&H88781222	DMUS_E_INVALID_PARAMCONTROLTRACK
&H88781226	DMUS_E_AUDIOPATHS_NOT_VALID
&H88781227	DMUS_E_AUDIOPATHS_IN_USE
&H88781228	DMUS_E_NO_AUDIOPATH_CONFIG
&H88781229	DMUS_E_AUDIOPATH_INACTIVE
&H8878122A	DMUS_E_AUDIOPATH_NOBUFFER
&H8878122B	DMUS_E_AUDIOPATH_NOPORT
&H8878122C	DMUS_E_NO_AUDIOPATH

DirectMusic Error Codes By Name

The following list describes all DirectMusic error codes. To find a constant from its value, see DirectMusic Error Codes By Number.

DMUS_E_ALL_TOOLS_FAILED
: The graph object was unable to load all tools, perhaps because of errors in the stream or because the tools are incorrectly registered on the client.

DMUS_E_ALL_TRACKS_FAILED
: The segment object was unable to load all tracks from the **IStream** object data, perhaps because of errors in the stream or because the tracks are incorrectly registered on the client.

DMUS_E_ALREADY_ACTIVATED
: The port has been activated, and the parameter cannot be changed.

DMUS_E_ALREADY_DOWNLOADED
: Buffer has already been downloaded.

DMUS_E_ALREADY_EXISTS
: The tool is already contained in the graph. You must create a new instance.

DMUS_E_ALREADY_INITED
: The object has already been initialized.

DMUS_E_ALREADY_LOADED
: The DLS collection is already open.

DMUS_E_ALREADY_SENT
: The message has already been sent.

DMUS_E_ALREADYCLOSED
The port is not open.

DMUS_E_ALREADYOPEN
The port was already opened.

DMUS_E_AUDIOPATH_INACTIVE
An audiopath is inactive, perhaps because the performance has been closed down.

DMUS_E_AUDIOPATH_NOBUFFER
The audiopath could not be created because a requested buffer could not be created.

DMUS_E_AUDIOPATH_NOPORT
The audiopath could not be used for playback because it lacked port assignments.

DMUS_E_AUDIOPATHS_IN_USE
The performance has set up audiopaths, so performance channels cannot be allocated.

DMUS_E_AUDIOPATHS_NOT_VALID
The performance is not using audiopaths.

DMUS_E_BADARTICULATION
Invalid articulation chunk in DLS collection.

DMUS_E_BADINSTRUMENT
Invalid instrument chunk in DLS collection.

DMUS_E_BADOFFSETTABLE
Offset table has errors.

DMUS_E_BADWAVE
Corrupted wave header.

DMUS_E_BADWAVELINK
Wave-link chunk in DLS collection points to an invalid wave.

DMUS_E_BUFFER_EMPTY
There is no data in the buffer.

DMUS_E_BUFFER_FULL
The specified number of bytes exceeds the maximum buffer size.

DMUS_E_BUFFERNOTAVAILABLE
The buffer is not available for download.

DMUS_E_BUFFERNOTSET
No buffer was prepared for the data.

DMUS_E_CANNOT_CONVERT
The requested conversion between music and MIDI values could not be made.

DMUS_E_CANNOT_FREE
The message could not be freed, either because it was not allocated or because it has already been freed.

DMUS_E_CANNOT_OPEN_PORT
The default system port could not be opened.

DMUS_E_CANNOTREAD

An error occurred when trying to read from the **IStream** object.

DMUS_E_CANNOTSEEK
The stream object does not support seeking.

DMUS_E_CANNOTWRITE
The stream object does not support writing.

DMUS_E_CHUNKNOTFOUND
A chunk with the specified header could not be found.

DMUS_E_DESCEND_CHUNK_FAIL
An attempt to descend into a chunk failed.

DMUS_E_DEVICE_IN_USE
Device is already in use (possibly by a non-DirectMusic client) and cannot be opened again.

DMUS_E_DMUSIC_RELEASED
Operation cannot be performed because the final instance of the DirectMusic object was released. Ports cannot be used after final release of the DirectMusic object.

DMUS_E_DRIVER_FAILED
An unexpected error was returned from a device driver, indicating possible failure of the driver or hardware.

DMUS_E_DSOUND_ALREADY_SET
A DirectSound object has already been set.

DMUS_E_DSOUND_NOT_SET
Port could not be created because no DirectSound object has been specified.

DMUS_E_FAIL
The method did not succeed.

DMUS_E_GET_UNSUPPORTED
Getting the parameter is not supported.

DMUS_E_INSUFFICIENTBUFFER
Buffer is not large enough for the requested operation.

DMUS_E_INVALID_BAND
File does not contain a valid band.

DMUS_E_INVALID_DOWNLOADID
Invalid download identifier was used in the process of creating a download buffer.

DMUS_E_INVALID_EVENT
The event either is not a valid MIDI message or makes use of running status, and cannot be packed into the buffer.

DMUS_E_INVALID_LYRICSTRACK
The file contains an invalid lyrics track.

DMUS_E_INVALID_PARAMCONTROLTRACK
The file contains an invalid parameter control track.

DMUS_E_INVALID_SEGMENTTRIGGERTRACK
The file contains an invalid segment trigger track.

DMUS_E_INVALID_TOOL_HDR
The stream object's data contains an invalid tool header and cannot be read by the graph object.

DMUS_E_INVALID_TRACK_HDR
The stream object's data contains an invalid track header and cannot be read by the segment object.

DMUS_E_INVALIDARG
An invalid value has been passed in a parameter.

DMUS_E_INVALIDBUFFER
Invalid DirectSound buffer was handed to port.

DMUS_E_INVALIDFILE
Not a valid file.

DMUS_E_INVALIDOFFSET
Wave chunks in the DLS collection file are at incorrect offsets.

DMUS_E_INVALIDPATCH
No instrument in the collection matches the patch number.

DMUS_E_INVALIDPOS
Error reading wave data from a DLS collection. Indicates a bad file.

DMUS_E_LOADER_BADPATH
The file path is invalid.

DMUS_E_LOADER_FAILEDCREATE
Object could not be found or created.

DMUS_E_LOADER_FAILEDOPEN
File open failed because the file does not exist or is locked.

DMUS_E_LOADER_FORMATNOTSUPPORTED
The object cannot be loaded because the data format is not supported.

DMUS_E_LOADER_NOCLASSID
No class identifier was supplied in the object description.

DMUS_E_LOADER_NOFILENAME
No file name was supplied in the object description.

DMUS_E_LOADER_OBJECTNOTFOUND
The object was not found.

DMUS_E_NO_AUDIOPATH
An attempt was made to play on a nonexistent audiopath.

DMUS_E_NO_AUDIOPATH_CONFIG
The segment does not contain an embedded audiopath configuration.

DMUS_E_NO_MASTER_CLOCK
There is no master clock in the performance, perhaps because the **DirectMusicPerformance8.InitAudio** method was not called.

DMUS_E_NOARTICULATION
Articulation missing from an instrument in the DLS collection.

DMUS_E_NOINTERFACE
No object interface is available.

DMUS_E_NOSYNTHSINK
No sink is connected to the synthesizer.

DMUS_E_NOT_DOWNLOADED_TO_PORT
The object cannot be unloaded because it is not present on the port.

DMUS_E_NOT_FOUND
The requested item is not contained by the object.

DMUS_E_NOT_INIT
A required object is not initialized or failed to initialize.

DMUS_E_NOT_LOADED
An attempt to use this object failed because it was not loaded.

DMUS_E_NOTADLSCOL
The object being loaded is not a valid DLS collection.

DMUS_E_NOTIMPL
The method is not implemented. This value can be returned if a driver does not support a feature necessary for the operation.

DMUS_E_NOTMONO
The wave chunk has more than one interleaved channel. DLS format requires mono.

DMUS_E_NOTPCM
Wave data is not in PCM format.

DMUS_E_OUT_OF_RANGE
The requested time is outside the range of the segment.

DMUS_E_OUTOFMEMORY
Insufficient memory to complete task.

DMUS_E_PORT_NOT_CAPTURE
The port is not a capture port.

DMUS_E_PORT_NOT_RENDER
Not an output port.

DMUS_E_PORTS_OPEN
The requested operation cannot be performed while there are instantiated ports in any process in the system.

DMUS_E_SEGMENT_INIT_FAILED
Segment initialization failed, probably because of a critical memory situation.

DMUS_E_SET_UNSUPPORTED
Setting the parameter is not supported.

DMUS_E_SYNTHACTIVE
The synthesizer has been activated, and the parameter cannot be changed.

DMUS_E_SYNTHINACTIVE
The synthesizer has not been activated and cannot process data.

DMUS_E_SYNTHNOTCONFIGURED
The synthesizer is not properly configured or opened.

DMUS_E_TIME_PAST
The time requested is in the past.

DMUS_E_TOOL_HDR_NOT_FIRST_CK
The stream object's data does not have a tool header as the first chunk and, therefore, cannot be read by the graph object.

DMUS_E_TRACK_HDR_NOT_FIRST_CK
The stream object's data does not have a track header as the first chunk and, therefore, cannot be read by the segment object.

DMUS_E_TRACK_NOT_FOUND
There is no track of the requested type.

DMUS_E_TYPE_DISABLED
Parameter is unavailable because it has been disabled.

DMUS_E_TYPE_UNSUPPORTED
Parameter is unsupported on this track.

DMUS_E_UNKNOWN_PROPERTY
The property set or item is not implemented by this port.

DMUS_E_UNKNOWNDOWNLOAD
The synthesizer does not support this type of download.

DMUS_E_UNSUPPORTED_STREAM
The stream does not contain data supported by the loading object.

DMUS_E_WAVEFORMATNOTSUPPORTED
Invalid buffer format was handed to the synthesizer sink.

DirectSound Visual Basic Reference

This section contains reference information for the API elements of Microsoft® DirectSound® for Microsoft® Visual Basic®. Reference material is divided into the following categories:

- DirectSound Classes
- DirectSound Types
- DirectSound Enumerations
- DirectSound Error Codes

DirectSound Classes

This section contains references for the following DirectSound classes:

- **DirectSound3DBuffer8**

- **DirectSound3DListener8**
- **DirectSound8**
- **DirectSoundCapture8**
- **DirectSoundCaptureBuffer8**
- **DirectSoundEnum8**
- **DirectSoundFXChorus8**
- **DirectSoundFXCompressor8**
- **DirectSoundFXDistortion8**
- **DirectSoundFXEcho8**
- **DirectSoundFXFlanger8**
- **DirectSoundFXGargle8**
- **DirectSoundFXI3DL2Reverb8**
- **DirectSoundFXParamEq8**
- **DirectSoundFXWavesReverb8**
- **DirectSoundPrimaryBuffer8**
- **DirectSoundSecondaryBuffer8**

DirectSound3DBuffer8

[#]Used to set and retrieve the position, orientation, and other parameters of a sound buffer in 3-D space.

A **DirectSound3DBuffer8** object is obtained by using the **DirectSoundSecondaryBuffer8.GetDirectSound3DBuffer** method on a secondary buffer created with 3-D capabilities. The object can also be obtained from an audiopath.

The methods of the **DirectSound3DBuffer8** class can be organized into the following groups.

Batch parameter manipulation	**GetAllParameters**
	SetAllParameters
Distance	**GetMaxDistance**
	GetMinDistance
	SetMaxDistance
	SetMinDistance
Obtaining objects	**GetDirectSoundBuffer**
Operation mode	**GetMode**
	SetMode

IDH_DirectSound3DBuffer8_dxaudio_vb

Position	**GetPosition**
	SetPosition
Sound projection cones	**GetConeAngles**
	GetConeOrientation
	GetConeOutsideVolume
	SetConeAngles
	SetConeOrientation
	SetConeOutsideVolume
Velocity	**GetVelocity**
	SetVelocity

See Also

Obtaining the 3-D Buffer Object

DirectSound3DBuffer8.GetAllParameters

#Retrieves information about the 3-D characteristics of a sound buffer.

object.**GetAllParameters(***buffer* **As DS3DBUFFER)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

buffer
DS3DBUFFER type that receives the description of the 3-D characteristics of the sound buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

DirectSound3DBuffer8.GetConeAngles

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

object.**GetConeAngles(** _
inCone **As Long,** _

IDH_DirectSound3DBuffer8.GetAllParameters_dxaudio_vb
IDH_DirectSound3DBuffer8.GetConeAngles_dxaudio_vb

outCone **As Long)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

inCone
Variable that receives the inside angle of the sound projection cone, in degrees.

outCone
Variable that receives the outside angle of the sound projection cone, in degrees.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

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

See Also

Sound Cones

DirectSound3DBuffer8.GetConeOrientation

[#]Retrieves the orientation of the sound projection cone for this sound buffer.

object.**GetConeOrientation(***orientation* **As D3DVECTOR)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

orientation
D3DVECTOR type that receives the current orientation of the sound projection cone. The vector information represents the center of the sound cone.

IDH_DirectSound3DBuffer8.GetConeOrientation_dxaudio_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

The values returned are not necessarily the same as those set by using the **DirectSound3DBuffer8.SetConeOrientation** method. DirectSound adjusts orientation vectors so that they have a magnitude of less than or equal to 1.0.

See Also

Sound Cones

DirectSound3DBuffer8.GetConeOutsideVolume

[#]Retrieves the current cone outside volume for this sound buffer.

object.**GetConeOutsideVolume() As Long**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

Return Values

Returns the current cone outside volume for this sound buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

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

See Also

IDH_DirectSound3DBuffer8.GetConeOutsideVolume_dxaudio_vb

Sound Cones

DirectSound3DBuffer8.GetDirectSoundBuffer

#Returns the secondary buffer from which the 3-D buffer was created.

object.**GetDirectSoundBuffer() As DirectSoundSecondaryBuffer8**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

Return Values

Returns a **DirectSoundSecondaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

Remarks

This method can be used to obtain a **DirectSoundSecondaryBuffer8** object when the **DirectSoundSecondaryBuffer8** object from which the **DirectSound3DBuffer8** object was created is no longer available.

DirectSound3DBuffer8.GetMaxDistance

#Retrieves the current maximum distance for this sound buffer, which is the distance beyond which the volume is no longer attenuated.

object.**GetMaxDistance() As Single**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

Return Values

Returns the current maximum distance setting.

IDH_DirectSound3DBuffer8.GetDirectSoundBuffer_dxaudio_vb
IDH_DirectSound3DBuffer8.GetMaxDistance_dxaudio_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

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

See Also

DirectSound3DBuffer8.GetMinDistance,
DirectSound3DBuffer8.SetMaxDistance, Minimum and Maximum Distances

DirectSound3DBuffer8.GetMinDistance

#Retrieves the current minimum distance for this sound buffer, which is the distance at which the sound is at maximum volume.

object.**GetMinDistance() As Single**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

Return Values

Returns the minimum distance setting.

Error Codes

If the method fails, an error is raised and **Err.Number** may be DSERR_INVALIDPARAM.

Remarks

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

See Also

DirectSound3DBuffer8.SetMinDistance,
DirectSound3DBuffer8.GetMaxDistance, Minimum and Maximum Distances

---

IDH_DirectSound3DBuffer8.GetMinDistance_dxaudio_vb

DirectSound3DBuffer8.GetMode

#Retrieves the current operation mode for 3-D sound processing.

object.**GetMode() As CONST_DS3DMODEFLAGS**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

Return Values

Returns the mode setting. This value is one of the constants of the **CONST_DS3DMODEFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

DirectSound3DBuffer8.GetPosition

#Retrieves the sound buffer's position, in distance units.

object.**GetPosition(***position* **As D3DVECTOR)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

position
D3DVECTOR type that receives the position of the sound buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

By default, distance units are meters, but you can change the units by using the **DirectSound3DListener8.SetDistanceFactor** method.

IDH_DirectSound3DBuffer8.GetMode_dxaudio_vb
IDH_DirectSound3DBuffer8.GetPosition_dxaudio_vb

DirectSound3DBuffer8.GetVelocity

#Retrieves the current velocity for this sound buffer.

object.**GetVelocity(***velocity* **As D3DVECTOR)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

velocity
D3DVECTOR type that receives the sound buffer's current velocity.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

Velocity is used for Doppler effects only. It does not move the buffer.

The default unit of measurement is meters per second, but you can change this by using the **DirectSound3DListener8.SetDistanceFactor** method.

See Also

DirectSound3DBuffer8.SetPosition, **DirectSound3DBuffer8.SetVelocity**, Doppler Factor

DirectSound3DBuffer8.SetAllParameters

#Sets all 3-D properties of a sound buffer.

object.**SetAllParameters(_**
buffer **As DS3DBUFFER, _**
applyFlag **As CONST_DS3DAPPLYFLAGS)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

buffer

IDH_DirectSound3DBuffer8.GetVelocity_dxaudio_vb
IDH_DirectSound3DBuffer8.SetAllParameters_dxaudio_vb

DS3DBUFFER type containing the information that describes the 3-D characteristics of the sound buffer.

applyFlag
Value that specifies when the setting should be applied. This value must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration,

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

DirectSound3DBuffer8.SetConeAngles

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

object.**SetConeAngles(_**
inCone **As Long, _**
outCone **As Long, _**
applyFlag **As CONST_DS3DAPPLYFLAGS)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

inCone
Inside angle of the sound projection cone, in degrees.

outCone
Outside angle of the sound projection cone, in degrees.

applyFlag
Value that specifies when the setting should be applied. This value must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

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

IDH_DirectSound3DBuffer8.SetConeAngles_dxaudio_vb

See Also

Sound Cones

DirectSound3DBuffer8.SetConeOrientation

#Sets the orientation of the sound projection cone for this sound buffer. This method has no effect unless the cone angle and cone volume factor have also been set.

```
object.SetConeOrientation( _
  x As Single, _
  y As Single, _
  z As Single, _
  applyFlag As CONST_DS3DAPPLYFLAGS)
```

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

x, y, z
Coordinates of the new sound cone orientation vector.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

See Also

Sound Cones

DirectSound3DBuffer8.SetConeOutsideVolume

#Sets the cone outside volume for this sound buffer.

```
object.SetConeOutsideVolume( _
  coneOutsideVolume As Long, _
  applyFlag As CONST_DS3DAPPLYFLAGS)
```

IDH_DirectSound3DBuffer8.SetConeOrientation_dxaudio_vb
IDH_DirectSound3DBuffer8.SetConeOutsideVolume_dxaudio_vb

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

coneOutsideVolume
Cone outside volume for this sound buffer, in hundredths of decibels.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

Volume levels are represented by attenuation. Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). The default value is DS3D_DEFAULTCONEOUTSIDEVOLUME (no attenuation).

See Also

DirectSoundSecondaryBuffer8.SetVolume, Sound Cones

DirectSound3DBuffer8.SetMaxDistance

[#]Sets the maximum distance for the buffer, which is the distance beyond which the sound is no longer attenuated.

```
object.SetMaxDistance( _
  maxDistance As Single, _
  applyFlag As CONST_DS3DAPPLYFLAGS)
```

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

maxDistance
Maximum distance value.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

IDH_DirectSound3DBuffer8.SetMaxDistance_dxaudio_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be DSERR_INVALIDPARAM.

Remarks

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

See Also

DirectSound3DBuffer8.GetMaxDistance,
DirectSound3DBuffer8.SetMinDistance, Minimum and Maximum Distances

DirectSound3DBuffer8.SetMinDistance

#Sets the minimum distance for the buffer, which is the distance at which the sound is at maximum volume.

object.**SetMinDistance(_**
minDistance **As Single, _**
applyFlag **As CONST_DS3DAPPLYFLAGS)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

minDistance
Minimum distance value.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

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

IDH_DirectSound3DBuffer8.SetMinDistance_dxaudio_vb

See Also

DirectSound3DBuffer8.SetMaxDistance, Minimum and Maximum Distances

DirectSound3DBuffer8.SetMode

#Sets the operation mode for 3-D sound processing.

object.**SetMode(** _
mode **As CONST_DS3DMODEFLAGS,** _
applyFlag **As CONST_DS3DAPPLYFLAGS)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

mode
One of the constants of the **CONST_DS3DMODEFLAGS** enumeration that specifies the 3-D sound processing mode to set.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

DirectSound3DBuffer8.SetPosition

#Sets the sound buffer's position, in distance units.

object.**SetPosition(** _
x **As Single,** _
y **As Single,** _
z **As Single,** _
applyFlag **As CONST_DS3DAPPLYFLAGS)**

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

x, y, z

IDH_DirectSound3DBuffer8.SetMode_dxaudio_vb
IDH_DirectSound3DBuffer8.SetPosition_dxaudio_vb

Values that specify the coordinates of the position vector. DirectSound may adjust these values to prevent floating-point overflow.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

By default, distance units are meters, but you can change the units by using the **DirectSound3DListener8.SetDistanceFactor** method.

DirectSound3DBuffer8.SetVelocity

#Sets the sound buffer's velocity.

```
object.SetVelocity( _
  x As Single, _
  y As Single, _
  z As Single, _
  applyFlag As CONST_DS3DAPPLYFLAGS)
```

Parts

object
Resolves to a **DirectSound3DBuffer8** object.

x, y, z
Values that specify the coordinates of the velocity vector. DirectSound may adjust these values to prevent floating-point overflow.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

IDH_DirectSound3DBuffer8.SetVelocity_dxaudio_vb

Velocity is used for Doppler effects only. It does not move the buffer.

The default unit of measurement is meters per second, but you can change this by using the **DirectSound3DListener8.SetDistanceFactor** method.

See Also

DirectSound3DBuffer8.SetPosition, **DirectSound3DBuffer8.GetVelocity**, Buffer Position and Velocity

DirectSound3DListener8

[#]Represents the position and orientation of the listener and controls global 3-D sound parameters.

An object of the **DirectSound3DListener8** class is obtained by using the **DirectSoundPrimaryBuffer8.GetDirectSound3DListener** method.

The methods of the **DirectSound3DListener8** class can be organized into the following groups.

Batch parameters	**GetAllParameters**
	SetAllParameters
Deferred settings	**CommitDeferredSettings**
Distance factor	**GetDistanceFactor**
	SetDistanceFactor
Doppler factor	**GetDopplerFactor**
	SetDopplerFactor
Obtaining objects	**GetDirectSoundBuffer**
Orientation	**GetOrientation**
	SetOrientation
Position	**GetPosition**
	SetPosition
Rolloff factor	**GetRolloffFactor**
	SetRolloffFactor
Velocity	**GetVelocity**
	SetVelocity

[#] IDH_DirectSound3DListener8_dxaudio_vb

DirectSound3DListener8.CommitDeferredSettings

#Commits all parameter changes made by passing the DS3D_DEFERRED flag to methods of **DirectSound3DBuffer8** and **DirectSound3DListener8**.

object.**CommitDeferredSettings()**

Parts

object
Resolves to a **DirectSound3DListener8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

See Also

Deferred Settings

DirectSound3DListener8.GetAllParameters

#Retrieves the state of the 3-D world and listener.

object.**GetAllParameters(***listener* **As DS3DLISTENER)**

Parts

object
Resolves to a **DirectSound3DListener8** object.
listener
DS3DLISTENER type that receives parameters.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

IDH_DirectSound3DListener8.CommitDeferredSettings_dxaudio_vb
IDH_DirectSound3DListener8.GetAllParameters_dxaudio_vb

See Also

DirectSound3DListener8.SetAllParameters

DirectSound3DListener8.GetDirectSoundBuffer

#Retrieves the primary buffer from which the listener was created.

object.**GetDirectSoundBuffer() As DirectSoundPrimaryBuffer8**

Parts

object
Resolves to a **DirectSound3DListener8** object.

Return Values

Returns a **DirectSoundPrimaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_OUTOFMEMORY.

DirectSound3DListener8.GetDistanceFactor

#Retrieves the distance factor, which is the number of meters in a vector unit.

object.**GetDistanceFactor() As Single**

Parts

object
Resolves to a **DirectSound3DListener8** object.

Return Values

Returns the distance factor.

IDH_DirectSound3DListener8.GetDirectSoundBuffer_dxaudio_vb
IDH_DirectSound3DListener8.GetDistanceFactor_dxaudio_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

See Also

DirectSound3DListener8.SetDistanceFactor, Distance Factor

DirectSound3DListener8.GetDopplerFactor

#Retrieves the Doppler effect factor.

object.**GetDopplerFactor() As Single**

Parts

object
Resolves to a **DirectSound3DListener8** object.

Return Values

Returns the Doppler factor.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

The Doppler factor has a range of DS3D_MINDOPPLERFACTOR (no Doppler effects) to DS3D_MAXDOPPLERFACTOR (10 times the Doppler effects found in the real world). The default value is DS3D_DEFAULTDOPPLERFACTOR (1.0).

See Also

DirectSound3DListener8.SetDopplerFactor, Doppler Factor

DirectSound3DListener8.GetOrientation

#Retrieves the listener's orientation in 3-D space.

IDH_DirectSound3DListener8.GetDopplerFactor_dxaudio_vb
IDH_DirectSound3DListener8.GetOrientation_dxaudio_vb

object.**GetOrientation(_**
orientFront **As D3DVECTOR, _**
orientTop **As D3DVECTOR)**

Parts

object
Resolves to a **DirectSound3DListener8** object.

orientFront
D3DVECTOR type that receives the listener's front orientation vector.

orientTop
D3DVECTOR type that receives the listener's top orientation vector.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

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

The values returned are not necessarily the same as those set by using the **DirectSound3DListener8.SetOrientation** method. DirectSound adjusts orientation vectors so that they are at right angles and have a magnitude of less than or equal to 1.0.

See Also

Listener Orientation

DirectSound3DListener8.GetPosition

[#]Retrieves the listener's position in 3-D space.

object.**GetPosition(***position* **As D3DVECTOR)**

Parts

object
Resolves to a **DirectSound3DListener8** object.

position
D3DVECTOR type that receives the listener's position vector.

IDH_DirectSound3DListener8.GetPosition_dxaudio_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to the value of DSERR_INVALIDPARAM.

See Also

DirectSound3DListener8.SetPosition, Listener Position and Velocity

DirectSound3DListener8.GetRolloffFactor

#Retrieves the rolloff factor, which controls the rate of attenuation over distance.

object.**GetRolloffFactor() As Single**

Parts

object
Resolves to a **DirectSound3DListener8** object.

Return Values

Returns the rolloff factor.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

The rolloff factor has a range of DS3D_MINROLLOFFFACTOR (no rolloff) to DS3D_MAXROLLOFFFACTOR (10 times the rolloff found in the real world). The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0).

See Also

DirectSound3DListener8.SetRolloffFactor, Rolloff Factor

DirectSound3DListener8.GetVelocity

#Retrieves the listener's velocity.

IDH_DirectSound3DListener8.GetRolloffFactor_dxaudio_vb
IDH_DirectSound3DListener8.GetVelocity_dxaudio_vb

object.**GetVelocity() As D3DVECTOR**

Parts

object
Resolves to a **DirectSound3DListener8** object.

Return Values

Returns a **D3DVECTOR** type containing the listener's velocity.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

The default velocity is (0,0,0).

Velocity is used only for Doppler effects. It does not move the listener.

See Also

DirectSound3DListener8.SetVelocity, Listener Position and Velocity

DirectSound3DListener8.SetAllParameters

#Sets all the properties of a 3-D listener and global sound parameters.

object.**SetAllParameters(_**
listener **As DS3DLISTENER, _**
applyFlag **As CONST_DS3DAPPLYFLAGS)**

Parts

object
Resolves to a **DirectSound3DListener8** object.

listener
DS3DLISTENER type that contains information describing all 3-D listener parameters.

applyFlag

IDH_DirectSound3DListener8.SetAllParameters_dxaudio_vb

Value that specifies when the setting should be applied. This value must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

See Also

DirectSound3DListener8.GetAllParameters

DirectSound3DListener8.SetDistanceFactor

[#]Sets the distance factor, which is the number of meters in a vector unit.

```
object.SetDistanceFactor( _
  distanceFactor As Single, _
  applyFlag As CONST_DS3DAPPLYFLAGS)
```

Parts

object
Resolves to a **DirectSound3DListener8** object.

distanceFactor
Distance factor.

applyFlag
Value that specifies when the setting should be applied. This value must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

See Also

DirectSound3DListener8.GetDistanceFactor, Distance Factor

IDH_DirectSound3DListener8.SetDistanceFactor_dxaudio_vb

DirectSound3DListener8.SetDopplerFactor

#Sets the Doppler effect factor.

object.**SetDopplerFactor(_**
dopplerFactor **As Single, _**
applyFlag **As CONST_DS3DAPPLYFLAGS)**

Parts

object
Resolves to a **DirectSound3DListener8** object.

dopplerFactor
Doppler factor value.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

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

See Also

DirectSound3DListener8.GetDopplerFactor, Doppler Factor

DirectSound3DListener8.SetOrientation

#Sets the listener's orientation in terms of a front vector and a top vector.

object.**SetOrientation(_**
xFront **As Single, _**
yFront **As Single, _**
zFront **As Single, _**

IDH_DirectSound3DListener8.SetDopplerFactor_dxaudio_vb
IDH_DirectSound3DListener8.SetOrientation_dxaudio_vb

xTop **As Single, _**
yTop **As Single, _**
zTop **As Single, _**
applyFlag **As CONST_DS3DAPPLYFLAGS)**

Parts

object
Resolves to a **DirectSound3DListener8** object.

xFront, yFront, zFront
Values that specify the coordinates of the front orientation vector.

xTop, yTop, zTop
Values that specify the coordinates of the top orientation vector.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

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

The top vector must be at right angles to the front vector. If necessary, DirectSound adjusts the front vector after setting the top vector.

See Also

DirectSound3DListener8.GetOrientation, Listener Orientation

DirectSound3DListener8.SetPosition

#Sets the listener's position, in distance units.

object.**SetPosition(_**
x **As Single, _**
y **As Single, _**
z **As Single, _**
applyFlag **As CONST_DS3DAPPLYFLAGS)**

IDH_DirectSound3DListener8.SetPosition_dxaudio_vb

Parts

object
Resolves to a **DirectSound3DListener8** object.

x, y, z
Values that specify the coordinates of the listener's position vector. DirectSound may adjust these values to prevent floating-point overflow.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

By default, these units are meters, but you can change this by calling the **DirectSound3DListener8.SetDistanceFactor** method.

See Also

DirectSound3DListener8.GetPosition, Listener Position and Velocity

DirectSound3DListener8.SetRolloffFactor

[#]Sets the rolloff factor, which determines the rate of attenuation over distance.

```
object.SetRolloffFactor( _
  rolloffFactor As Single, _
  applyFlag As CONST_DS3DAPPLYFLAGS)
```

Parts

object
Resolves to a **DirectSound3DListener8** object.

rolloffFactor
Rolloff factor.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

IDH_DirectSound3DListener8.SetRolloffFactor_dxaudio_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

The rolloff factor has a range of DS3D_MINROLLOFFFACTOR (no rolloff) to DS3D_MAXROLLOFFFACTOR (defined as 10 times the rolloff found in the real world). The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0).

See Also

DirectSound3DListener8.GetRolloffFactor, Rolloff Factor

DirectSound3DListener8.SetVelocity

#Sets the listener's velocity.

```
object.SetVelocity( _
  x As Single, _
  y As Single, _
  z As Single, _
  applyFlag As CONST_DS3DAPPLYFLAGS)
```

Parts

object
Resolves to a **DirectSound3DListener8** object.

x, y, z
Values that specify the coordinates of the listener's velocity vector. DirectSound may adjust these values to prevent floating-point overflow.

applyFlag
Value that specifies when the setting should be applied. Must be one of the constants of the **CONST_DS3DAPPLYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

The default velocity is (0,0,0).

IDH_DirectSound3DListener8.SetVelocity_dxaudio_vb

Velocity is used only for Doppler effects. It does not move the listener. To change the listener's position, use the **DirectSound3DListener8.SetPosition** method.

See Also

DirectSound3DListener8.GetVelocity, Listener Position and Velocity

DirectSound8

#Represents the DirectSound playback system. Applications use the methods of the **DirectSound8** class to create buffer objects and set up the audio environment.

An object of this class is obtained by using the **DirectX8.DirectSoundCreate** function.

The methods of the **DirectSound8** class can be organized into the following groups.

Initialization	**SetCooperativeLevel**
Buffer creation	**CreatePrimarySoundBuffer**
	CreateSoundBuffer
	CreateSoundBufferFromFile
	CreateSoundBufferFromResource
	DuplicateSoundBuffer
Device capabilities	**GetCaps**
Speaker configuration	**GetSpeakerConfig**
	SetSpeakerConfig

DirectSound8.CreatePrimarySoundBuffer

#Creates a primary buffer object that can be used to change the format of the audio output and to obtain a **DirectSound3DListener8** object.

```
object.CreatePrimarySoundBuffer( _
  bufferDesc As DSBUFFERDESC _
) As DirectSoundPrimaryBuffer8
```

Parts

object
Resolves to a **DirectSound8** object.

bufferDesc

IDH_DirectSound8_dxaudio_vb
IDH_DirectSound8.CreatePrimarySoundBuffer_dxaudio_vb

DSBUFFERDESC type that specifies the description of the sound buffer to create. The **lBufferBytes** member must be 0, and **lFlags** must contain DSBCAPS_PRIMARYBUFFER.

Return Values

Returns a **DirectSoundPrimaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following error values.

DSERR_ALLOCATED
DSERR_BADFORMAT
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

DirectSound8.CreateSoundBuffer

[#]Creates a secondary buffer to mix audio samples into the primary buffer.

```
object.CreateSoundBuffer( _
  bufferDesc As DSBUFFERDESC _
) As DirectSoundSecondaryBuffer8
```

Parts

object
Resolves to a **DirectSound8** object.

bufferDesc
DSBUFFERDESC type that specifies the description of the sound buffer to create.

Return Values

Returns a **DirectSoundSecondaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following error values.

IDH_DirectSound8.CreateSoundBuffer_dxaudio_vb

DSERR_ALLOCATED
DSERR_BADFORMAT

DSERR_BUFFERTOOSMALL
DSERR_CONTROLUNAVAIL
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

Remarks

DirectSound does not initialize the contents of the buffer, and the application cannot assume that it contains silence.

If an attempt is made to create a buffer with the DSBCAPS_LOCHARDWARE flag on a system where hardware acceleration is not available, the method fails with either DSERR_CONTROLUNAVAIL or DSERR_INVALIDCALL, depending on the operating system.

See Also

DirectSound8.CreateSoundBufferFromFile, **DirectSound8.CreateSoundBufferFromResource**, **DirectSound8.DuplicateSoundBuffer**

DirectSound8.CreateSoundBufferFromFile

[#]Creates a secondary buffer to mix audio samples into the primary buffer, and loads data from a file into the buffer.

object.**CreateSoundBufferFromFile(_**
  filename **As String, _**
  bufferDesc **As DSBUFFERDESC _**
) As DirectSoundSecondaryBuffer8

Parts

object
    Resolves to a **DirectSound8** object.

IDH_DirectSound8.CreateSoundBufferFromFile_dxaudio_vb

filename
Name of the wave file to load into the created buffer.

bufferDesc
DSBUFFERDESC type that specifies the description of the sound buffer to create. The **fxFormat** and **lBufferBytes** members are ignored, as the method determines the format and size from the data itself.

Return Values

Returns a **DirectSoundSecondaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following error values.

DSERR_ALLOCATED
DSERR_BADFORMAT
DSERR_BUFFERTOOSMALL
DSERR_CONTROLUNAVAIL
DSERR_INVALIDCALL
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

Remarks

If an attempt is made to create a buffer with the DSBCAPS_LOCHARDWARE flag on a system where hardware acceleration is not available, the method fails with DSERR_CONTROLUNAVAIL on Windows 95 and Windows 98. On Windows 2000, it returns DSERR_INVALIDCALL.

See Also

DirectSound8.CreateSoundBuffer,
DirectSound8.CreateSoundBufferFromResource,
DirectSound8.DuplicateSoundBuffer

DirectSound8.CreateSoundBufferFromResource

[#]Creates a secondary buffer to mix audio samples into the primary buffer, and loads data from a resource into the buffer. The sound resource must be of type WAV or WAVE.

```
object.CreateSoundBufferFromResource( _
  dll As String, _
  resourceName As String, _
  bufferDesc As DSBUFFERDESC _
) As DirectSoundSecondaryBuffer8
```

Parts

object
Resolves to a **DirectSound8** object.

dll
Name of the DLL that contains the resource, or vbNullString if the resource is contained in the executable file.

resourceName
Name of the resource to load into the created sound buffer.

bufferDesc
DSBUFFERDESC type that specifies the description of the sound buffer to create. The **fxFormat** and **lBufferBytes** members are ignored, as the method determines the format and size from the data itself.

Return Values

Returns a **DirectSoundSecondaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following error values.

DSERR_ALLOCATED
DSERR_BADFORMAT
DSERR_BUFFERTOOSMALL
DSERR_CONTROLUNAVAIL
DSERR_INVALIDCALL
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

IDH_DirectSound8.CreateSoundBufferFromResource_dxaudio_vb

DSERR_UNSUPPORTED

Remarks

If an attempt is made to create a buffer with the DSBCAPS_LOCHARDWARE flag on a system where hardware acceleration is not available, the method fails with DSERR_CONTROLUNAVAIL on Windows 95 and Windows 98. On Windows 2000, it returns DSERR_INVALIDCALL.

See Also

DirectSound8.CreateSoundBuffer, **DirectSound8.CreateSoundBufferFromFile**, **DirectSound8.DuplicateSoundBuffer**

DirectSound8.DuplicateSoundBuffer

#Creates a new secondary buffer that duplicates or shares the original buffer's memory.

```
object.DuplicateSoundBuffer( _
  original As DirectSoundSecondaryBuffer8 _
) As DirectSoundSecondaryBuffer8
```

Parts

object
Resolves to a **DirectSound8** object.

original
DirectSoundSecondaryBuffer8 object to duplicate.

Return Values

Returns a **DirectSoundSecondaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following error values.

DSERR_ALLOCATED
DSERR_INVALIDCALL
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

IDH_DirectSound8.DuplicateSoundBuffer_dxaudio_vb

Remarks

This method is not valid for buffers created with the DSBCAPS_CTRLFX flag.

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

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

Although duplicate buffers often share the same memory, this is not always the case. Changes in the data stored in one buffer are not necessarily reflected in its duplicates.

See Also

DirectSound8.CreateSoundBuffer, **DirectSound8.CreateSoundBufferFromFile**, **DirectSound8.CreateSoundBufferFromResource**

DirectSound8.GetCaps

[#]Retrieves the capabilities of the hardware device represented by the **DirectSound8** object.

object.**GetCaps(***caps* **As DSCAPS)**

Parts

object
Resolves to a **DirectSound8** object.

caps
DSCAPS type that receives the capabilities of this sound device.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following error values.

DSERR_GENERIC
DSERR_UNINITIALIZED

Remarks

Information retrieved in the **DSCAPS** type describes the maximum capabilities of the sound device and those currently available, such as the number of hardware mixing

---

IDH_DirectSound8.GetCaps_dxaudio_vb

channels and the amount of on-board sound memory. You can use this information to fine-tune performance and optimize resource allocation.

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

See Also

DirectX8.DirectSoundCreate

DirectSound8.GetSpeakerConfig

#Retrieves the speaker configuration for this **DirectSound8** object.

object.**GetSpeakerConfig() As CONST_DSSPEAKERFLAGS**

Parts

object
Resolves to a **DirectSound8** object.

Return Values

Returns a **Long** value containing one or more of the constants of the **CONST_DSSPEAKERFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following error values.

DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

The speaker geometry is set by default to zero, which states that the speaker geometry has not been defined. The geometry must be explicitly set by using the **DirectSound8.SetSpeakerConfig** method and passing one of the speaker geometry flags with the DSSPEAKER_STEREO flag.

See Also

DirectSound8.SetSpeakerConfig

IDH_DirectSound8.GetSpeakerConfig_dxaudio_vb

DirectSound8.SetCooperativeLevel

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

object.**SetCooperativeLevel(_**
hwnd **As Long, _**
level **As CONST_DSSCLFLAGS)**

Parts

object
Resolves to a **DirectSound8** object.

hwnd
Window handle to the application. See Remarks.

level
Requested priority level. Must be a constant of the **CONST_DSSCLFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following error values.

DSERR_ALLOCATED
DSERR_INVALIDPARAM
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

Remarks

The application must set the cooperative level by calling this method before its buffers can be played. The recommended cooperative level is DSSCL_PRIORITY.

The *hwnd* parameter should be the top-level application window handle.

See Also

Cooperative Levels

DirectSound8.SetSpeakerConfig

#Specifies the speaker configuration of the **DirectSound8** object.

object.**SetSpeakerConfig(_**

IDH_DirectSound8.SetCooperativeLevel_dxaudio_vb
IDH_DirectSound8.SetSpeakerConfig_dxaudio_vb

speakerConfig **As CONST_DSSPEAKERFLAGS)**

Parts

object
Resolves to a **DirectSound8** object.

speakerConfig
Speaker configuration of the **DirectSound8** object. Must be one or more of the constants of the **CONST_DSSPEAKERFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following error values.

DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

The speaker geometry is set by default to zero, which indicates that the speaker geometry has not been defined. The geometry must be explicitly set by passing one of the speaker geometry flags combined with the DSSPEAKER_STEREO flag.

See Also

DirectSound8.GetSpeakerConfig, Speaker Configuration

DirectSoundCapture8

#Represents a sound capture device. The methods of the class are used to create sound capture buffers and query the capabilities of the device.

A **DirectSoundCapture8** object is obtained by using the **DirectX8.DirectSoundCaptureCreate** method.

This reference section gives information on the following methods of the **DirectSoundCapture8** class.

Buffer creation	**CreateCaptureBuffer**
Capabilities	**GetCaps**

See Also

Capturing Waves

IDH_DirectSoundCapture8_dxaudio_vb

DirectSoundCapture8.CreateCaptureBuffer

#Creates a capture buffer.

object.**CreateCaptureBuffer(_**
bufferDesc **As DSCBUFFERDESC _**
) As DirectSoundCaptureBuffer8

Parts

object
Resolves to a **DirectSoundCapture8** object.

bufferDesc
DSCBUFFERDESC type that specifies values for the capture buffer being created.

Return Values

Returns a **DirectSoundCaptureBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_INVALIDPARAM
DSERR_BADFORMAT
DSERR_GENERIC
DSERR_NODRIVER
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

Remarks

On Microsoft® Windows® 95, Windows® 98, and Windows® 2000, each capture device supports a single buffer.

DirectSoundCapture8.GetCaps

#Obtains the capabilities of the capture device.

object.**GetCaps(***caps* **As DSCCAPS)**

IDH_DirectSoundCapture8.CreateCaptureBuffer_dxaudio_vb
IDH_DirectSoundCapture8.GetCaps_dxaudio_vb

Parts

object
Resolves to a **DirectSoundCapture8** object.

caps
DSCCAPS type that receives the capabilities of the capture device.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_INVALIDPARAM
DSERR_UNSUPPORTED
DSERR_NODRIVER
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

DirectSoundCaptureBuffer8

[#]Represents a buffer used to capture sounds.

An object of the **DirectSoundCaptureBuffer8** class is obtained by calling the **DirectSoundCapture8.CreateCaptureBuffer** method.

The methods of the **DirectSoundCaptureBuffer8** class may be grouped as follows:

Information	**GetCaps**
	GetCurrentPosition
	GetFormat
	GetStatus
Buffer data	**ReadBuffer**
	WriteBuffer
Notification	**SetNotificationPositions**
Capture management	**Start**
	Stop

See Also

Capturing Waves

IDH_DirectSoundCaptureBuffer8_dxaudio_vb

DirectSoundCaptureBuffer8.GetCaps

#Returns the capabilities of the buffer.

object.**GetCaps(***caps* **As DSCBCAPS)**

Parts

object
Resolves to a **DirectSoundCaptureBuffer8** object.

caps
DSCBCAPS type that receives the capabilities of the capture buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_INVALIDPARAM
DSERR_UNSUPPORTED
DSERR_OUTOFMEMORY

DirectSoundCaptureBuffer8.GetCurrentPosition

#Retrieves the capture and read cursors.

object.**GetCurrentPosition(***cursors* **As DSCURSORS)**

Parts

object
Resolves to a **DirectSoundCaptureBuffer8** object.

cursors
DSCURSORS type that receives the position of the capture cursor and the read cursor. Each value is an offset from the start of the buffer, in bytes. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_INVALIDPARAM

IDH_DirectSoundCaptureBuffer8.GetCaps_dxaudio_vb
IDH_DirectSoundCaptureBuffer8.GetCurrentPosition_dxaudio_vb

DSERR_NODRIVER
DSERR_OUTOFMEMORY

Remarks

The capture cursor, **DSCURSORS.lPlay** indicates the point to which data is being captured. The read cursor, **DSCURSORS.lWrite**, is behind the capture cursor and indicates the point up to which data can safely be read. The data after **lWrite**, up to and including **lPlay**, is not necessarily valid data, because of buffering.

See Also

Capture Buffer Information

DirectSoundCaptureBuffer8.GetFormat

#Retrieves the format of the capture buffer.

object.**GetFormat(***waveformat* **As WAVEFORMATEX)**

Parts

object
Resolves to a **DirectSoundCaptureBuffer8** object.

waveformat
WAVEFORMATEX type that receives a description of the capture buffer format.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

DirectSoundCaptureBuffer8.GetStatus

#Retrieves the current status of the capture buffer.

object.**GetStatus() As CONST_DSCBSTATUSFLAGS**

Parts

object

IDH_DirectSoundCaptureBuffer8.GetFormat_dxaudio_vb
IDH_DirectSoundCaptureBuffer8.GetStatus_dxaudio_vb

Resolves to a **DirectSoundCaptureBuffer8** object.

Return Values

Returns a **Long** value containing one or more of the constants of the **CONST_DSCBSTATUSFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

DirectSoundCaptureBuffer8.ReadBuffer

#Reads a segment of the sound capture buffer and stores the data in an application buffer.

```
object.ReadBuffer( _
  start As Long, _
  size As Long, _
  buffer As Any, _
  flags As CONST_DSCBLOCKFLAGS)
```

Parts

object
Resolves to a **DirectSoundCaptureBuffer8** object.

start
Offset, in bytes, from the start of the buffer to where the read begins.

size
Size, in bytes, of the portion of the buffer to read. The sound buffer is conceptually circular, so this value can be greater than the number of bytes between *start* and the end of the buffer. This parameter is ignored if the DSCBLOCK_ENTIREBUFFER constant is set in the *flags* argument.

buffer
Variable to hold the data read. Typically this is an array of bytes, and you pass the first element of the array.

flags
One of the constants of the **CONST_DSCBLOCKFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

IDH_DirectSoundCaptureBuffer8.ReadBuffer_dxaudio_vb

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

See Also

DirectSoundCaptureBuffer8.GetCurrentPosition

DirectSoundCaptureBuffer8.SetNotificationPositions

#Sets points in the buffer where notification events are to be signaled. An event is signaled when the read cursor reaches a notification position.

object.**SetNotificationPositions(** _
nElements **As Long**, _
psa() **As DSBPOSITIONNOTIFY)**

Parts

object
Resolves to a **DirectSoundCaptureBuffer8** object.

nElements
Number of elements in the *psa* array.

psa
Array of **DSBPOSITIONNOTIFY** types that specify the notification positions and associated events.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

See Also

DirectSoundCaptureBuffer8.GetCurrentPosition

IDH_DirectSoundCaptureBuffer8.SetNotificationPositions_dxaudio_vb

DirectSoundCaptureBuffer8.Start

#Begins capturing data into the buffer. If the capture buffer is already capturing data, the method has no effect.

object.**Start(***flags* **As CONST_DSCBSTARTFLAGS)**

Parts

object
Resolves to a **DirectSoundCaptureBuffer8** object.

flags
Flags that specify the behavior for the capture buffer when capturing sound data. Can be one of the constants of the **CONST_DSCBSTARTFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_INVALIDPARAM
DSERR_NODRIVER
DSERR_OUTOFMEMORY

DirectSoundCaptureBuffer8.Stop

#Stops capturing data. If the capture buffer is already stopped, the method has no effect.

object.**Stop()**

Parts

object
Resolves to a **DirectSoundCaptureBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values

DSERR_NODRIVER

IDH_DirectSoundCaptureBuffer8.Start_dxaudio_vb
IDH_DirectSoundCaptureBuffer8.Stop_dxaudio_vb

DSERR_OUTOFMEMORY

DirectSoundCaptureBuffer8.WriteBuffer

#Writes data to the capture buffer. Most applications do not use this method, as data is written to the buffer in response to the **DirectSoundCaptureBuffer8.Start** method.

object.**WriteBuffer(_**
start **As Long, _**
size **As Long, _**
buffer **As Any, _**
flags **As CONST_DSCBLOCKFLAGS)**

Parts

object
Resolves to a **DirectSoundCaptureBuffer8** object.

start
Offset, in bytes, from the start of the buffer to where the write begins.

size
Size, in bytes, of the portion of the buffer to read. The sound buffer is conceptually circular, so this value can be greater than the number of bytes between *start* and the end of the buffer. This parameter is ignored if DSCBLOCK_ENTIREBUFFER is set in the *flags* parameter.

buffer
Variable containing the data to write to the capture buffer. Typically this is an array of bytes, and you pass the first element of the array.

flags
One of the constants from the **CONST_DSCBLOCKFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

See Also

DirectSoundCaptureBuffer8.GetCurrentPosition

IDH_DirectSoundCaptureBuffer8.WriteBuffer_dxaudio_vb

DirectSoundEnum8

#An object of the **DirectSoundEnum8** class represents a collection of DirectSound-capable devices. The object is obtained by using the **DirectX8.GetDSEnum** or **DirectX8.GetDSCaptureEnum** method.

This class has the following methods:

Information	**GetCount**
	GetDescription
	GetGuid
	GetName

See Also

Enumeration of Sound Devices, Enumeration of Capture Devices

DirectSoundEnum8.GetCount

#Retrieves the number of devices in the enumeration.

object.**GetCount() As Long**

Parts

object
Resolves to a **DirectSoundEnum8** object.

Return Values

Returns the number of entries in the enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectSoundEnum8.GetDescription

#Retrieves the friendly name of a device in the enumeration.

object.**GetDescription(***index* **As Long) As String**

IDH_DirectSoundEnum8_dxaudio_vb
IDH_DirectSoundEnum8.GetCount_dxaudio_vb
IDH_DirectSoundEnum8.GetDescription_dxaudio_vb

Parts

object
Resolves to a **DirectSoundEnum8** object.

index
Index of the device in the enumeration.

Return Values

Returns the description of the device supplied by the driver.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

DirectSoundEnum8.GetGuid

#Retrieves the unique identifier of a device in the enumeration.

object.**GetGuid(***index* **As Long) As String**

Parts

object
Resolves to a **DirectSoundEnum8** object.

index
Index of the device in the enumeration.

Return Values

The method returns the GUID of the device in string form.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

See Also

Using GUIDs

DirectSoundEnum8.GetName

#Returns the name of a device driver in the enumeration.

IDH_DirectSoundEnum8.GetGuid_dxaudio_vb
IDH_DirectSoundEnum8.GetName_dxaudio_vb

object.**GetName(***index* **As Long) As String**

object
Resolves to a **DirectSoundEnum8** object.

index
Index of the device in the enumeration.

Return Values

Returns the name of the device driver. This is not the friendly name of the device, which is returned by **DirectSoundEnum8.GetDescription**.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

DirectSoundFXChorus8

#Used to set and retrieve effect parameters on a buffer that supports chorus.

An object of this class is obtained by calling **DirectSoundSecondaryBuffer8.GetObjectInPath** on the buffer that supports the effect.

The **DirectSoundFXChorus8** class has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

See Also

DirectSoundSecondaryBuffer8.SetFX

DirectSoundFXChorus8.GetAllParameters

#Retrieves the chorus parameters of a buffer.

object.**GetAllParameters() As DSFXCHORUS**

Parts

object
Resolves to a **DirectSoundFXChorus8** object.

IDH_DirectSoundFXChorus8_dxaudio_vb
IDH_DirectSoundFXChorus8.GetAllParameters_dxaudio_vb

Return Values

Returns a **DSFXCHORUS** type that contains the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXChorus8.SetAllParameters

#Sets the amplitude modulation parameters of a buffer.

object.**SetAllParameters(***params* **As DSFXCHORUS)**

Parts

object
Resolves to a **DirectSoundFXChorus8** object.

params
DSFXCHORUS type that specifies the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXCompressor8

#Used to set and retrieve effect parameters on a buffer that supports compressor effects.

An object of this class is obtained by calling **DirectSoundSecondaryBuffer8.GetObjectInPath** on the buffer that supports the effect.

The **DirectSoundFXCompressor8** class has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

IDH_DirectSoundFXChorus8.SetAllParameters_dxaudio_vb
IDH_DirectSoundFXCompressor8_dxaudio_vb

See Also

DirectSoundSecondaryBuffer8.SetFX, Compression

DirectSoundFXCompressor8.GetAllParameters

#Retrieves the compressor parameters of a buffer.

object.**GetAllParameters() As DSFXCOMPRESSOR**

Parts

object
Resolves to a **DirectSoundFXCompressor8** object.

Return Values

Returns a **DSFXCOMPRESSOR** type that contains the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXCompressor8.SetAllParameters

#Sets the compression parameters of a buffer.

object.**SetAllParameters(***params* **As DSFXCOMPRESSOR)**

Parts

object
Resolves to a **DirectSoundFXCompressor8** object.

params
DSFXCOMPRESSOR type that specifies the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

IDH_DirectSoundFXCompressor8.GetAllParameters_dxaudio_vb
IDH_DirectSoundFXCompressor8.SetAllParameters_dxaudio_vb

DirectSoundFXDistortion8

#Used to set and retrieve effect parameters on a buffer that supports distortion effects.

An object of this class is obtained by calling **DirectSoundSecondaryBuffer8.GetObjectInPath** on the buffer that supports the effect.

The **DirectSoundFXDistortion8** class has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

See Also

DirectSoundSecondaryBuffer8.SetFX

DirectSoundFXDistortion8.GetAllParameters

#Retrieves the distortion parameters of a buffer.

*object.***GetAllParameters() As DSFXDISTORTION**

Parts

object
Resolves to a **DirectSoundFXDistortion8** object.

Return Values

Returns a **DSFXDISTORTION** type that contains the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXDistortion8.SetAllParameters

#Sets the distortion parameters of a buffer.

*object.***SetAllParameters(***params* **As DSFXDISTORTION)**

IDH_DirectSoundFXDistortion8_dxaudio_vb
IDH_DirectSoundFXDistortion8.GetAllParameters_dxaudio_vb
IDH_DirectSoundFXDistortion8.SetAllParameters_dxaudio_vb

Parts

object
Resolves to a **DirectSoundFXDistortion8** object.

params
DSFXDISTORTION type that specifies the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXEcho8

#Used to set and retrieve effect parameters on a buffer that supports echo effects.

An object of this class is obtained by calling **DirectSoundSecondaryBuffer8.GetObjectInPath** on the buffer that supports the effect.

The **DirectSoundFXEcho8** class has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

See Also

DirectSoundSecondaryBuffer8.SetFX

DirectSoundFXEcho8.GetAllParameters

#Retrieves the echo parameters of a buffer.

*object.***GetAllParameters() As DSFXECHO**

Parts

object
Resolves to a **DirectSoundFXEcho8** object.

Return Values

Returns a **DSFXECHO** type that contains the parameters of the effect.

IDH_DirectSoundFXEcho8_dxaudio_vb
IDH_DirectSoundFXEcho8.GetAllParameters_dxaudio_vb

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXEcho8.SetAllParameters

#Sets the echo parameters of a buffer.

object.**SetAllParameters(***params* **As DSFXECHO)**

Parts

object
Resolves to a **DirectSoundFXEcho8** object.

params
DSFXECHO type that specifies the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXFlanger8

#Used to set and retrieve effect parameters on a buffer that supports flanger effects.

An object of this class is obtained by calling **DirectSoundSecondaryBuffer8.GetObjectInPath** on the buffer that supports the effect.

The **DirectSoundFXFlanger8** class has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

See Also

DirectSoundSecondaryBuffer8.SetFX

DirectSoundFXFlanger8.GetAllParameters

#Retrieves the flanger parameters of a buffer.

IDH_DirectSoundFXEcho8.SetAllParameters_dxaudio_vb
IDH_DirectSoundFXFlanger8_dxaudio_vb
IDH_DirectSoundFXFlanger8.GetAllParameters_dxaudio_vb

*object.***GetAllParameters() As DSFXFLANGER**

Parts

object
Resolves to a **DirectSoundFXFlanger8** object.

Return Values

Returns a **DSFXFLANGER** type that contains the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXFlanger8.SetAllParameters

#Sets the flanger parameters of a buffer.

*object.***SetAllParameters(***params* **As DSFXFLANGER)**

Parts

object
Resolves to a **DirectSoundFXFlanger8** object.

params
DSFXFLANGER type that specifies the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXGargle8

#Used to set and retrieve effect parameters on a buffer that supports amplitude modulation.

An object of this class is obtained by calling **DirectSoundSecondaryBuffer8.GetObjectInPath** on the buffer that supports the effect.

The **DirectSoundFXGargle8** class has the following methods.

IDH_DirectSoundFXFlanger8.SetAllParameters_dxaudio_vb
IDH_DirectSoundFXGargle8_dxaudio_vb

Parameters	**GetAllParameters**
	SetAllParameters

See Also

DirectSoundSecondaryBuffer8.SetFX

DirectSoundFXGargle8.GetAllParameters

#Retrieves the amplitude modulation parameters of a buffer.

*object.***GetAllParameters() As DSFXGARGLE**

Parts

object
Resolves to a **DirectSoundFXGargle8** object.

Return Values

Returns a **DSFXGARGLE** type that contains the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXGargle8.SetAllParameters

#Sets the amplitude modulation parameters of a buffer.

*object.***SetAllParameters(***params* **As DSFXGARGLE)**

Parts

object
Resolves to a **DirectSoundFXGargle8** object.

params
DSFXGARGLE type that specifies the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

IDH_DirectSoundFXGargle8.GetAllParameters_dxaudio_vb
IDH_DirectSoundFXGargle8.SetAllParameters_dxaudio_vb

DirectSoundFXI3DL2Reverb8

#Used to set and retrieve effect parameters on a buffer that supports I3DL2 (Interactive 3D Audio Level 2) reverberation effects.

An object of this class is obtained by calling **DirectSoundSecondaryBuffer8.GetObjectInPath** on the buffer that supports the effect.

The **DirectSoundFXI3DL2Reverb8** class has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters
Presets	**GetPreset**
	SetPreset
Quality	**GetQuality**
	SetQuality

See Also

DirectSoundSecondaryBuffer8.SetFX

DirectSoundFXI3DL2Reverb8.GetAllParameters

#Retrieves the I3DL2 (Interactive 3D Audio Level 2) reverberation parameters of a buffer.

object.**GetAllParameters() As DSFXI3DL2REVERB**

Parts

object
Resolves to a **DirectSoundFXI3DL2Reverb8** object.

Return Values

Returns a **DSFXI3DL2REVERB** type that contains the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

IDH_DirectSoundFXI3DL2Reverb8_dxaudio_vb
IDH_DirectSoundFXI3DL2Reverb8.GetAllParameters_dxaudio_vb

DirectSoundFXI3DL2Reverb8.GetPreset

#Retrieves an identifier for standard reverberation parameters of a buffer.

object.**GetPreset() As Long**

Parts

object
Resolves to a **DirectSoundFXI3DL2Reverb8** object.

Return Values

Returns a preset identifier from the **CONST_DSFX_I3DL2_ENVIRONMENT_PRESETS** enumeration. See Dsound.h for the default parameters associated with each preset.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include the following:

DSERR_INVALIDPARAM
DSERR_INVALIDCALL

DirectSoundFXI3DL2Reverb8.GetQuality

#Retrieves the quality of the environmental reverberation effect.

object.**GetQuality() As Long**

Parts

object
Resolves to a **DirectSoundFXI3DL2Reverb8** object.

Return Values

Returns the quality of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

IDH_DirectSoundFXI3DL2Reverb8.GetPreset_dxaudio_vb
IDH_DirectSoundFXI3DL2Reverb8.GetQuality_dxaudio_vb

See Also

DirectSoundFXI3DL2Reverb8.SetQuality

DirectSoundFXI3DL2Reverb8.SetAllParameters

#Sets the I3DL2 (Interactive 3D Audio Level 2) reverberation parameters of a buffer.

object.**SetAllParameters(***params* **As DSFXI3DL2REVERB)**

Parts

object
Resolves to a **DirectSoundFXI3DL2Reverb8** object.

params
DSFXI3DL2REVERB type that specifies the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXI3DL2Reverb8.SetPreset

#Sets standard reverberation parameters of a buffer.

object.**SetPreset(***lPreset* **As Long)**

Parts

object
Resolves to a **DirectSoundFXI3DL2Reverb8** object.

lPreset
Constant from the **CONST_DSFX_I3DL2_ENVIRONMENT_PRESETS** enumeration that specifies a set of default parameters. See Dsound.h for the parameters associated with each constant.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

IDH_DirectSoundFXI3DL2Reverb8.SetAllParameters_dxaudio_vb
IDH_DirectSoundFXI3DL2Reverb8.SetPreset_dxaudio_vb

DirectSoundFXI3DL2Reverb8.SetQuality

#Sets the quality of the environmental reverberation effect. Higher values produce better quality at the expense of processing time.

object.**SetQuality(_**
lQuality **As CONST_DSFX_I3DL2REVERB_QUALITY)**

Parts

object
Resolves to a **DirectSoundFXI3DL2Reverb8** object.

lQuality
Value that specifies the quality of the effect. Minimum and maximum values are defined in the **CONST_DSFX_I3DL2REVERB_QUALITY** enumeration. The default value is 2.

Return Values

Returns the quality of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

See Also

DirectSoundFXI3DL2Reverb8.GetQuality

DirectSoundFXParamEq8

#Used to set and retrieve effect parameters on a buffer that supports parametric equalizer effects.

An object of this class is obtained by calling **DirectSoundSecondaryBuffer8.GetObjectInPath** on the buffer that supports the effect.

The **DirectSoundFXParameq8** class has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

IDH_DirectSoundFXI3DL2Reverb8.SetQuality_dxaudio_vb
IDH_DirectSoundFXParamEq8_dxaudio_vb

See Also

DirectSoundSecondaryBuffer8.SetFX

DirectSoundFXParamEq8.GetAllParameters

#Retrieves the parametric equalizer parameters of a buffer.

object.**GetAllParameters() As DSFXPARAMEQ**

Parts

object
Resolves to a **DirectSoundFXParamEq8** object.

Return Values

Returns a **DSFXPARAMEQ** type that contains the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundFXParamEq8.SetAllParameters

#Sets the parametric equalizer parameters of a buffer.

object.**SetAllParameters(***params* **As DSFXPARAMEQ)**

Parts

object
Resolves to a **DirectSoundFXParamEq8** object.

params
DSFXPARAMEQ type that specifies the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

IDH_DirectSoundFXParamEq8.GetAllParameters_dxaudio_vb
IDH_DirectSoundFXParamEq8.SetAllParameters_dxaudio_vb

DirectSoundFXWavesReverb8

#Used to set and retrieve effect parameters on a buffer that that supports wave reverberation.

An object of this class is obtained by calling **DirectSoundSecondaryBuffer8.GetObjectInPath** on the buffer that supports the effect.

The **DirectSoundFXWavesReverb8** class has the following methods.

Parameters	**GetAllParameters**
	SetAllParameters

Remarks

The Waves reverberation DMO is based on the Waves MaxxVerb technology, which is licenced to Microsoft.

See Also

DirectSoundSecondaryBuffer8.SetFX

DirectSoundFXWavesReverb8.GetAllParameters

#Retrieves the wave reverberation parameters of a buffer.

object.**GetAllParameters() As DSFXWAVESREVERB**

Parts

object
Resolves to a **DirectSoundFXWavesReverb8** object.

Return Values

Returns a **DSFXWAVESREVERB** type that contains the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

IDH_DirectSoundFXWavesReverb8_dxaudio_vb
IDH_DirectSoundFXWavesReverb8.GetAllParameters_dxaudio_vb

DirectSoundFXWavesReverb8.SetAllParameters

#Sets the wave reverberation parameters of a buffer.

object.**SetAllParameters(***params* **As DSFXWAVESREVERB)**

Parts

object
Resolves to a **DirectSoundFXWavesReverb8** object.

params
DSFXWAVESREVERB type that specifies the parameters of the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_INVALIDPARAM.

DirectSoundPrimaryBuffer8

#Represents the primary sound buffer.

An object of the **DirectSoundPrimaryBuffer8** class can be obtained by using the **DirectSound8.CreatePrimarySoundBuffer** method. Typically a primary buffer object is used to set the output format and to obtain a 3-D listener that controls global sound parameters. Specialized applications can handle the mixing of data into the primary buffer, but normally this is handled by DirectSound when secondary buffers are played.

The methods of this class can be organized into the following groups.

Data streaming	**Play**
	ReadBuffer
	Stop
	WriteBuffer
Parameters	**GetCaps**
	GetCurrentPosition
	GetDirectSound3DListener
	GetFormat
	GetPan
	GetStatus

IDH_DirectSoundFXWavesReverb8.SetAllParameters_dxaudio_vb
IDH_DirectSoundPrimaryBuffer8_dxaudio_vb

	GetVolume
	SetFormat
	SetPan
	SetVolume
Miscellaneous	**Restore**

See Also

DirectSoundSecondaryBuffer8

DirectSoundPrimaryBuffer8.GetCaps

#Retrieves the capabilities of the buffer.

object.**GetCaps(***caps* **As DSBCAPS)**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

caps
DSBCAPS type that receives the capabilities of the buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

DirectSoundPrimaryBuffer8.GetCurrentPosition

#Retrieves the position of the play and write cursors.

object.**GetCurrentPosition(***cursors* **As DSCURSORS)**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

cursors

IDH_DirectSoundPrimaryBuffer8.GetCaps_dxaudio_vb
IDH_DirectSoundPrimaryBuffer8.GetCurrentPosition_dxaudio_vb

DSCURSORS type that receives the offset of the play cursor and the write cursor, in bytes.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values.

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The write cursor is the point in the buffer ahead of which it is safe to write data to the buffer. Data should not be written to the part of the buffer after the play cursor and before the write cursor. For more information, see Play and Write Cursors.

DirectSoundPrimaryBuffer8.GetDirectSound3DListener

[#]Returns a listener object, which represents the person listening to the sound. The listener object can be used to set global sound parameters.

object.**GetDirectSound3DListener() As DirectSound3DListener8**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

Return Values

Returns a **DirectSound3DListener8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

Remarks

The method succeeds only if the primary buffer was created with the DSBCAPS_CTRL3D flag.

IDH_DirectSoundPrimaryBuffer8.GetDirectSound3DListener_dxaudio_vb

DirectSoundPrimaryBuffer8.GetFormat

#Retrieves a description of the wave format of the buffer.

object.**GetFormat(***format* **As WAVEFORMATEX)**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

format
WAVEFORMATEX type that receives a description of the wave format of the buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

See Also

DirectSoundPrimaryBuffer8.SetFormat

DirectSoundPrimaryBuffer8.GetPan

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

object.**GetPan() As Long**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

Return Values

Returns the relative mix between the left and right speakers. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

IDH_DirectSoundPrimaryBuffer8.GetFormat_dxaudio_vb
IDH_DirectSoundPrimaryBuffer8.GetPan_dxaudio_vb

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The return value is in hundredths of a decibel (dB), in the range of DSBPAN_LEFT (-10,000) to DSBPAN_RIGHT (10,000). The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER (0), meaning that both channels are at full volume (attenuated by 0 decibels).

At any setting other than DSBPAN_CENTER, one channel is at full volume and the other is attenuated. For example, a value 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.

The pan control acts cumulatively with the volume control.

See Also

DirectSoundPrimaryBuffer8.GetVolume, **DirectSoundPrimaryBuffer8.SetPan**, **DirectSoundPrimaryBuffer8.SetVolume**

DirectSoundPrimaryBuffer8.GetStatus

[#]Retrieves the status of the sound buffer.

object.**GetStatus() As CONST_DSBSTATUSFLAGS**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

Return Values

Returns the status of the sound buffer. The status can be zero, indicating that the buffer is neither lost nor playing, or a combination of the constants of the **CONST_DSBSTATUSFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

IDH_DirectSoundPrimaryBuffer8.GetStatus_dxaudio_vb

DirectSoundPrimaryBuffer8.GetVolume

#Retrieves the volume for the sound buffer.

object.**GetVolume() As Long**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

Return Values

Returns the volume of the buffer. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The volume is specified in hundredths of decibels (dB) of attenuation. Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are defined as 0 and -10,000, respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the sound. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for practical purposes, is silence.

See Also

DirectSoundPrimaryBuffer8.SetVolume, **DirectSoundPrimaryBuffer8.GetPan**

DirectSoundPrimaryBuffer8.Play

#Causes the primary buffer to play.

object.**Play(***flags* **As CONST_DSBPLAYFLAGS)**

IDH_DirectSoundPrimaryBuffer8.GetVolume_dxaudio_vb
IDH_DirectSoundPrimaryBuffer8.Play_dxaudio_vb

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

flags
Must be DSBPLAY_LOOPING.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

See Also

DirectSoundPrimaryBuffer8.Stop

DirectSoundPrimaryBuffer8.ReadBuffer

#Reads a segment of the buffer and stores the data in an application buffer. Most applications do not use this method, as the primary buffer streams its own data to the sound device in response to the **DirectSoundPrimaryBuffer8.Play** method.

```
object.ReadBuffer( _
  start As Long, _
  size As Long, _
  buffer As Any, _
  flags As CONST_DSBLOCKFLAGS)
```

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

start
Offset, in bytes, from the start of the buffer to where the read begins. This parameter is ignored if DSBLOCK_FROMWRITECURSOR is specified in the *flags* parameter.

size
Size, in bytes, of the portion of the buffer to read. The sound buffer is conceptually circular, so this value can be greater than the number of bytes

IDH_DirectSoundPrimaryBuffer8.ReadBuffer_dxaudio_vb

between *start* and the end of the buffer. This member is ignored if DSBLOCK_ENTIREBUFFER is specified in the *flags* parameter.

buffer
Variable to hold the data read. Typically this is an array of bytes, and you pass the first element of the array.

flags
One of the constants of the **CONST_DSBLOCKFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

See Also

DirectSoundPrimaryBuffer8.WriteBuffer

DirectSoundPrimaryBuffer8.Restore

[#]Restores the memory allocation for a lost sound buffer.

object.**Restore()**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_PRIOLEVELNEEDED

IDH_DirectSoundPrimaryBuffer8.Restore_dxaudio_vb

Remarks

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

The application can receive notification that a buffer is lost when it specifies that buffer in a call to the **DirectSoundSecondaryBuffer8.Play** method. This method raises the DSERR_BUFFERLOST error to indicate a lost buffer. The **DirectSoundSecondaryBuffer8.GetStatus** method can also be used to retrieve the status of the sound buffer and test for the DSBSTATUS_BUFFERLOST flag.

See Also

DirectSoundPrimaryBuffer8.GetStatus

DirectSoundPrimaryBuffer8.SetFormat

#Sets the format of the buffer. Whenever this application gains the input focus, DirectSound sets the primary buffer to the specified format.

object.**SetFormat(***format* **As WAVEFORMATEX)**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

format
WAVEFORMATEX type that specifies the new format for the primary sound buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_BADFORMAT
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY
DSERR_PRIOLEVELNEEDED

Remarks

The format of the primary buffer should be set before secondary buffers are created.

IDH_DirectSoundPrimaryBuffer8.SetFormat_dxaudio_vb

The method fails if the application has the DSSCL_NORMAL cooperative level.

If the application is using DirectSound at the DSSCL_WRITEPRIMARY cooperative level, the buffer must be stopped before **SetFormat** is called. If the format is not supported, the method fails.

If the cooperative level is DSSCL_PRIORITY, DirectSound stops the primary buffer, changes the format, and restarts the buffer. The method succeeds even if the hardware does not support the requested format; DirectSound sets the buffer to the closest supported format. To determine whether this has happened, an application can call the **DirectSoundPrimaryBuffer8.GetFormat** method for the primary buffer and compare the result with the format that was requested with the **SetFormat** method.

This method is not available for secondary sound buffers. If a new format is required, the application must create a new DirectSoundBuffer object.

See Also

Matching Buffer Formats

DirectSoundPrimaryBuffer8.SetPan

[#]Specifies the relative volume between the left and right channels.

object.**SetPan(***pan* **As Long)**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

pan
Relative volume between the left and right channels.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

IDH_DirectSoundPrimaryBuffer8.SetPan_dxaudio_vb

Remarks

The return value is in hundredths of a decibel (dB), in the range of DSBPAN_LEFT (-10,000) to DSBPAN_RIGHT (10,000). The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER (0), meaning that both channels are at full volume (attenuated by 0 decibels).

At any setting other than DSBPAN_CENTER, one channel is at full volume and the other is attenuated. For example, a value 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.

The pan control acts cumulatively with the volume control.

See Also

DirectSoundPrimaryBuffer8.GetPan, **DirectSoundPrimaryBuffer8.SetVolume**

DirectSoundPrimaryBuffer8.SetVolume

[#]Changes the sound volume of the buffer.

object.**SetVolume(***volume* **As Long)**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

volume
Volume requested for this sound buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

IDH_DirectSoundPrimaryBuffer8.SetVolume_dxaudio_vb

Remarks

The volume is specified in hundredths of decibels (dB) of attenuation. Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are defined as 0 and -10,000 respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the sound. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for practical purposes, is silence.

See Also

DirectSoundPrimaryBuffer8.GetVolume, **DirectSoundPrimaryBuffer8.SetPan**

DirectSoundPrimaryBuffer8.Stop

[#]Causes the sound buffer to stop playing.

object.**Stop()**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

if an application has the DSSCL_WRITEPRIMARY level, this method stops the primary buffer and reset the play cursor to the beginning of the buffer. This is necessary because the primary buffers on most sound cards can play only from the beginning of the buffer.

However, if **DirectSoundPrimaryBuffer8.Stop** is called on a primary buffer and the application has a cooperative level other than DSSCL_WRITEPRIMARY, this method simply reverses the effects of **DirectSoundPrimaryBuffer8.Play**. It configures the primary buffer to stop if no secondary buffers are playing. If other buffers are playing in this or other applications, the primary buffer will not actually stop until they are stopped. This method is useful because playing the primary buffer

---

IDH_DirectSoundPrimaryBuffer8.Stop_dxaudio_vb

consumes processing overhead even if the buffer is playing sound data with the amplitude of 0 decibels.

DirectSoundPrimaryBuffer8.WriteBuffer

[#]Writes audio data to the buffer.

It is not recommended that applications write directly to the primary buffer. Calling **DirectSoundSecondaryBuffer8.Play** causes data from the secondary buffer to be mixed into the primary buffer.

object.**WriteBuffer(** _
start **As Long,** _
size **As Long,** _
buffer **As Any,** _
flags **As CONST_DSBLOCKFLAGS)**

Parts

object
Resolves to a **DirectSoundPrimaryBuffer8** object.

start
Offset, in bytes, from the start of the buffer to where the read begins. This parameter is ignored if DSBLOCK_FROMWRITECURSOR constant is specified in the *flags* parameter.

size
Size, in bytes, of the portion of the buffer to read. The sound buffer is conceptually circular, so this value can be greater than the number of bytes between *start* and the end of the buffer. This parameter is ignored if DSBLOCK_ENTIREBUFFER constant is specified in the *flags* parameter.

buffer
Variable that contains the data to write to the sound buffer. Typically this is an array of bytes, and you pass the first element of the array.

flags
One of the constants of the **CONST_DSBLOCKFLAGS** enumeration specifying how to write the data.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM

IDH_DirectSoundPrimaryBuffer8.WriteBuffer_dxaudio_vb

DSERR_PRIOLEVELNEEDED

DirectSoundSecondaryBuffer8

#An object of the **DirectSoundSecondaryBuffer8** class represents a secondary play buffer. Sounds from secondary buffers are mixed into the primary buffer.

The object can be obtained by using the **DirectSound8.CreateSoundBuffer**, **DirectSound8.CreateSoundBufferFromFile**, or **DirectSound8.CreateSoundBufferFromResource** method. In applications that use audiopaths, a secondary buffer can be retrieved by using the **DirectMusicAudioPath8.GetObjectInPath** method.

The **DirectSoundSecondaryBuffer8** methods can be organized into the following groups.

Effects	**SetFX**
Information	**GetCaps**
	GetFormat
	GetStatus
Memory management	**AcquireResources**
	Restore
Miscellaneous	**GetObjectInPath**
	SaveToFile
Obtaining objects	**GetDirectSound3DBuffer**
Play management	**GetCurrentPosition**
	Play
	ReadBuffer
	SetCurrentPosition
	SetNotificationPositions
	Stop
	WriteBuffer
Sound management	**GetFrequency**
	GetPan
	GetVolume
	SetFrequency
	SetPan
	SetVolume

IDH_DirectSoundSecondaryBuffer8_dxaudio_vb

The following table shows which methods are supported for buffer objects obtained from an audiopath; that is, buffers created by a DirectMusic performance. Mix-in buffers accept sends from other buffers. All other buffers in the audiopath are sink-in buffers, which means that they accept data only from the synthesizer sink.

DirectSoundSecondaryBuffer8 method	*Mix-in*	*Sink-in*
AcquireResources		
GetCaps	Yes	Yes
GetCurrentPosition		
GetDirectSound3D Buffer	Yes	Yes
GetFormat	Yes	Yes
GetFrequency		
GetObjectInPath	Yes	Yes
GetPan	Yes	Yes
GetStatus	Yes	Yes
GetVolume	Yes	Yes
Play	Yes	
ReadBuffer		
Restore		
SaveToFile		
SetCurrentPosition		
SetFrequency		
SetFX	Yes	Yes
SetNotificationPositions		
SetPan	Yes	Yes
SetVolume	Yes	Yes
Stop	Yes	
WriteBuffer		

See Also

DirectSoundPrimaryBuffer8, DirectSound Buffers

DirectSoundSecondaryBuffer8.AcquireResources

[#]Allocates resources for a buffer created with the DSBCAPS_LOCDEFER flag.

```
object.AcquireResources( _
  lFlags As Long, _
```

[#] IDH_DirectSoundSecondaryBuffer8.AcquireResources_dxaudio_vb

lEffectsResults() **As Long)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

lFlags
One or more flags from the **CONST_DSBPLAYFLAGS** enumeration, other than DSBPLAY_LOOPING, specifying how resources are to be allocated.

lEffectsResults
Array of variables of type **Long** that receives information about the effects associated with the buffer. The number of elements must be equal to the number of effects previously set on the buffer by using **DirectSoundSecondaryBuffer8.SetFX**. For each effect, one of the following constants from the **CONST_DSOUNDFX** enumeration is returned.

DSFXR_FAILED
No effect was created because resources weren't available.

DSFXR_LOCHARDWARE
Effect is instantiated in hardware.

DSFXR_LOCSOFTWARE
Effect is instantiated in software.

DSFXR_PRESENT
The effect is available but was not created because the method failed for some other reason.

DSFXR_UNKNOWN
No effect was created because the effect isn't registered on the system.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include the following:

DSERR_CONTROLUNAVAIL
DSERR_INVALIDCALL
DSERR_INVALIDPARAM

Remarks

Normally, buffers created with DSBCAPS_LOCDEFER are not given resources until **DirectSoundSecondaryBuffer8.Play** is called. **DirectSoundSecondaryBuffer8.AcquireResources** can be used to allocate resources for deferred buffers. By doing so, the application can retrieve information about effects processing and set effect parameters before the buffer is played.

If the method fails, the value for each effect in *lEffectsResults* is either DSFXF_PRESENT or DSFXR_UNKNOWN. Check these values to determine which effects caused the failure.

A buffer with acquired resources that is not yet playing is not a candidate for premature termination by the voice management flags passed to the **Play** method.

If **AcquireResources** is called on a buffer on which it has already been called, the status of the effects is returned but no additional resources are allocated.

Resources are released when playback is stopped.

If an attempt is made to acquire resources for a buffer with the DSBCAPS_LOCHARDWARE flag on a system where hardware acceleration is not available, the method fails with either DSERR_CONTROLUNAVAIL or DSERR_INVALIDCALL, depending on the operating system.

DirectSoundSecondaryBuffer8.GetCaps

[#]Retrieves the capabilities of the buffer.

object.**GetCaps(***caps* **As DSBCAPS)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

caps
DSBCAPS type that receives the capabilities of the buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

Remarks

The **DSBCAPS** type is similar to the **DSBUFFERDESC** type passed to the **DirectSound8.CreateSoundBuffer** method, with additional information. This information can include the buffer's location, either in hardware or software, and performance measures.

The flags in the **lFlags** member of the **DSBCAPS** type are the same flags used by the **DSBUFFERDESC** type. The only difference is that in the **DSBCAPS** type, either DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE is specified according to the location of the buffer memory. In the **DSBUFFERDESC** type, these

[#] IDH_DirectSoundSecondaryBuffer8.GetCaps_dxaudio_vb

flags are optional and, depending on which flag is specified, force the buffer to be located in either hardware or software.

DirectSoundSecondaryBuffer8.GetCurrentPosition

[#]Retrieves the position of the play and write cursors.

object.**GetCurrentPosition(***cursors* **As DSCURSORS)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

cursors
DSCURSORS type that receives the offset of the play cursor and the write cursor, in bytes.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values.

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The write cursor is the point in the buffer ahead of which it is safe to write data to the buffer. Data should not be written to the part of the buffer after the play cursor and before the write cursor. For more information, see Play and Write Cursors.

See Also

DirectSoundSecondaryBuffer8.SetCurrentPosition

DirectSoundSecondaryBuffer8.GetDirectSound3DBuffer

[#]Retrieves an object that can be used to manipulate 3-D effects on the buffer.

object.**GetDirectSound3DBuffer() As DirectSound3DBuffer8**

IDH_DirectSoundSecondaryBuffer8.GetCurrentPosition_dxaudio_vb
IDH_DirectSoundSecondaryBuffer8.GetDirectSound3DBuffer_dxaudio_vb

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

Return Values

Returns a **DirectSound3DBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to E_NOINTERFACE or DSERR_OUTOFMEMORY.

Remarks

The method succeeds only on buffers that have been created with the DSBCAPS_CTRL3D flag.

DirectSoundSecondaryBuffer8.GetFormat

[#]Retrieves a description of the wave format of the buffer.

object.**GetFormat(***format* **As WAVEFORMATEX)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

format
WAVEFORMATEX type that receives a description of the wave format of the buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DSERR_INVALIDPARAM.

IDH_DirectSoundSecondaryBuffer8.GetFormat_dxaudio_vb

DirectSoundSecondaryBuffer8.GetFrequency

#Retrieves the frequency at which the buffer is playing.

object.**GetFrequency() As Long**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

Return Values

Returns the frequency in hertz, or samples per second.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The frequency value is in the range from DSBFREQUENCY_MIN (100) to DSBFREQUENCY_MAX (100,000).

See Also

DirectSoundSecondaryBuffer8.SetFrequency

DirectSoundSecondaryBuffer8.GetObjectInPath

#Retrieves an object in the audio path of the buffer. Typically this method is used to retrieve an object representing a sound effect in a buffer or DMO.

object.**GetObjectinPath(** _
guidFX **As String,** _
lIndex **As Long,** _

IDH_DirectSoundSecondaryBuffer8.GetFrequency_dxaudio_vb
IDH_DirectSoundSecondaryBuffer8.GetObjectInPath_dxaudio_vb

iidInterface **As String _**
) As Unknown

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

guidFX
String that specifies the unique identifier for the effect class. See Remarks. Can be one of the members of the AUDIOSTRINGCONSTANTS module that begins with DSFX, or a GUID in string form. See Remarks.

lIndex
Value of type **Long** that specifies the index of the object within objects of that class in the path. This is the index of the object in the array of effects passed to **DirectSoundSecondaryBuffer8.SetFX**. This is not necessarily the actual position of the object in the effects chain, because some effects might not have been created.

iidInterface
String that specifies the unique identifier of the class of the returned object. Can be one of the members of the **AUDIOSTRINGCONSTANTS** module that begins with IID, or a GUID in string form. See Remarks.

Return Values

Returns an object representing the effect.

Error Codes

If the method fails, an error is raised. Possible values for **Err.Number** include DSERR_OBJECTNOTFOUND.

Remarks

The following table shows the standard effects objects that can be retrieved from a buffer on which the effects have been set.

guidFX	***iidInterface***	**Returned object**
DSFX_STANDARD_CHORUS	IID_DirectSoundFXChorus	**DirectSoundFXChorus8**
DSFX_STANDARD_COMPRESSOR	IID_DirectSoundFXCompressor	**DirectSoundFXCompressor8**
DSFX_STANDARD_DISTORTION	IID_DirectSoundFXDistortion	**DirectSoundFXDistortion8**
DSFX_STANDARD_ECHO	IID_DirectSoundFXEcho	**DirectSoundFXEcho8**
DSFX_STANDARD_FLAN	IID_DirectSoundFXFl	**DirectSoundFXFlanger8**

GER	anger	
DSFX_STANDARD_GARGLE	IID_DirectSoundFXGargle	**DirectSoundFXGargle8**
DSFX_STANDARD_I3DL2REVERB	IID_DirectSoundFXI3DL2REVERB	**DirectSoundFXI3DL2Reverb8**
DSFX_STANDARD_PARAMEQ	IID_DirectSoundFXParamEq	**DirectSoundFXParamEq8**
DSFX_STANDARD_WAVES_REVERB	IID_DirectSoundFXWavesReverb	**DirectSoundFXWavesReverb8**

Any DMO that has been set on a buffer by using **DirectSoundSecondaryBuffer8.SetFX** can be retrieved, even it has not been allocated resources.

See Also

DirectSoundSecondaryBuffer8.SetFX, Using GUIDs

DirectSoundSecondaryBuffer8.GetPan

[#]Retrieves the relative volume between the left and right audio channels.

object.**GetPan() As Long**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

Return Values

Returns the relative attenuation of the channels. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

IDH_DirectSoundSecondaryBuffer8.GetPan_dxaudio_vb

Remarks

The return value is in hundredths of a decibel (dB), in the range of DSBPAN_LEFT (-10,000) to DSBPAN_RIGHT (10,000). The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER (0), meaning that both channels are at full volume (attenuated by 0 decibels).

At any setting other than DSBPAN_CENTER, one channel is at full volume and the other is attenuated. For example, a value 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.

The pan control acts cumulatively with the volume control.

See Also

DirectSoundSecondaryBuffer8.GetVolume,
DirectSoundSecondaryBuffer8.SetPan,
DirectSoundSecondaryBuffer8.SetVolume

DirectSoundSecondaryBuffer8.GetStatus

[#]Retrieves the status of the sound buffer.

object.**GetStatus() As CONST_DSBSTATUSFLAGS**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

Return Values

Returns the status of the sound buffer. The status can be zero or one or more of the constants of the **CONST_DSBSTATUSFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

IDH_DirectSoundSecondaryBuffer8.GetStatus_dxaudio_vb

DirectSoundSecondaryBuffer8.GetVolume

#Retrieves the volume for the sound buffer.

object.**GetVolume() As Long**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

Return Values

Returns the attenuation of the buffer, in hundredths of a decibel. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The return value is in the range from DSBVOLUME_MAX through DSBVOLUME_MIN. These constants are defined as 0 and -10,000 respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the sound. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for practical purposes, is silence.

See Also

DirectSoundSecondaryBuffer8.SetVolume,
DirectSoundSecondaryBuffer8.GetPan

DirectSoundSecondaryBuffer8.Play

#Causes the sound buffer to play from the play cursor.

object.**Play(***flags* **As CONST_DSBPLAYFLAGS)**

IDH_DirectSoundSecondaryBuffer8.GetVolume_dxaudio_vb
IDH_DirectSoundSecondaryBuffer8.Play_dxaudio_vb

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

flags
One or more of the constants of the **CONST_DSBPLAYFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

Before this method can be used the first time, the application must call the **DirectSound8.SetCooperativeLevel** method.

If the buffer specified in the method is already playing, the call succeeds and the buffer continues to play. However, the flags defined in the most recent call supersede the flags defined in previous calls.

If the method is called with a voice allocation or voice management flag set on a buffer that was not created with the DSBCAPS_LOCDEFER flag, the call fails with DSERR_INVALIDPARAM.

DSBPLAY_TERMINATEBY_TIME and DSBPLAY_TERMINATEBY_DISTANCE cannot be combined, but either can be combined with DSBPLAY_TERMINATEBY_PRIORITY, in which case the DSBPLAY_TERMINATEBY_TIME or DSBPLAY_TERMINATEBY_DISTANCE flag is used to determine which buffer should be terminated in the event of a priority tie.

The following table shows the behavior of the method under various combinations of the voice allocation and voice management flags when no free hardware voices are available.

DSBPLAY_LOCHARDWARE	**Neither DSBPLAY_LOCHARDWARE nor DSBPLAY_LOCSOFTWARE**	**DSBPLAY_LOCSOFTWARE**

DSBPLAY_TERMINATEBY_TIME	Nonlooping sound with least time left to play is terminated and the new sound plays on the released voice.	Nonlooping sound with least time left to play is terminated and the new sound plays on the released voice.	New sound plays in software.
DSBPLAY_TERMINATEBY_DISTANCE	If any sounds currently playing in hardware are beyond their maximum distance and have the DSBCAPS_MUTE3DATMAXDISTANCE flag set, one of them is terminated and the new sound plays in hardware. Otherwise, the call fails.	If any sounds currently playing in hardware are beyond their maximum distance and have the DSBCAPS_MUTE3DATMAXDISTANCE flag set, one of them is terminated and the new sound plays in hardware. Otherwise, the new sound plays in software.	New sound plays in software.
DSBPLAY_TERMINATEBY_PRIORITY	If the new sound's priority is higher than or equal to that of any sound currently playing in hardware, one of the lowest-priority sounds is terminated and the new sound plays in hardware. Otherwise, the call fails.	If the new sound's priority is higher than or equal to that of any sound currently playing in hardware, one of the lowest-priority sounds is terminated and the new sound plays in hardware. Otherwise, the new sound plays in software.	New sound plays in software.

See Also

Dynamic Voice Management, Effect Parameters

DirectSoundSecondaryBuffer8.ReadBuffer

[#]Reads a segment of the sound buffer and stores the data in an application buffer.

Most applications do not use this method, because secondary buffers automatically send their data to the primary buffer in response to the **DirectSoundSecondaryBuffer8.Play** method.

IDH_DirectSoundSecondaryBuffer8.ReadBuffer_dxaudio_vb

object.**ReadBuffer(_**
start **As Long, _**
size **As Long, _**
buffer **As Any, _**
flags **As CONST_DSBLOCKFLAGS)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

start
Offset, in bytes, from the start of the buffer to where the read begins. This parameter is ignored if DSBLOCK_FROMWRITECURSOR is specified in the *flags* parameter.

size
Size, in bytes, of the portion of the buffer to read. The sound buffer is conceptually circular, so this value can be greater than the number of bytes between *start* and the end of the buffer. This member is ignored if DSBLOCK_ENTIREBUFFER is specified in the *flags* parameter.

buffer
Variable that receives the data. Typically this is an array of bytes, and you pass the first element of the array.

flags
One of the constants of the **CONST_DSBLOCKFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

DirectSoundSecondaryBuffer8.Restore

[#]Restores the memory allocation for a lost sound buffer.

object.**Restore()**

IDH_DirectSoundSecondaryBuffer8.Restore_dxaudio_vb

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_PRIOLEVELNEEDED

Remarks

If the application does not have the input focus, **DirectSoundSecondaryBuffer8.Restore** might not succeed. For example, if the application with the input focus has the DSSCL_WRITEPRIMARY cooperative level, no other application will be able to restore its buffers.

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

The application can receive notification that a buffer is lost when it specifies that buffer in a call to the **DirectSoundSecondaryBuffer8.WriteBuffer** or **DirectSoundSecondaryBuffer8.Play** method. These methods return DSERR_BUFFERLOST to indicate a lost buffer. The **DirectSoundSecondaryBuffer8.GetStatus** method can also be used to retrieve the status of the sound buffer and test for the DSBSTATUS_BUFFERLOST flag.

See Also

DirectSoundSecondaryBuffer8.GetStatus, Buffer Management

DirectSoundSecondaryBuffer8.SaveToFile

[#]Saves the contents of the buffer to a wave file.

object.**SaveToFile(***filename* **As String)**

IDH_DirectSoundSecondaryBuffer8.SaveToFile_dxaudio_vb

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

filename
Name of the file where the sound buffer will be saved. If a file of that name exists, it is overwritten.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_INVALIDCALL
DSERR_INVALIDPARAM

See Also

Writing to a Wave File

DirectSoundSecondaryBuffer8.SetCurrentPosition

[#]Sets the position of the play cursor.

object.**SetCurrentPosition(***newPosition* **As Long)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

newPosition
Offset of the play cursor, in bytes, from the beginning of the buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values

DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

IDH_DirectSoundSecondaryBuffer8.SetCurrentPosition_dxaudio_vb

Remarks

If the buffer is playing, the cursor immediately moves to the new position and play continues from that point. If it is not playing, playback will begin from the new position the next time the **DirectSoundSecondaryBuffer8.Play** method is called.

See Also

DirectSoundSecondaryBuffer8.GetCurrentPosition,Play and Write Cursors

DirectSoundSecondaryBuffer8.SetFrequency

#Sets the frequency at which the audio samples are played.

object.**SetFrequency(***frequency* **As Long)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

frequency
New frequency, in hertz, at which to play the audio samples. The value must be in the range DSBFREQUENCY_MIN (100) to DSBFREQUENCY_MAX (100,000).

If the value is DSBFREQUENCY_ORIGINAL, the frequency is reset to the default value in the buffer format. This format is specified in the **DirectSound8.CreateSoundBuffer** method.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

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

IDH_DirectSoundSecondaryBuffer8.SetFrequency_dxaudio_vb

See Also

DirectSoundSecondaryBuffer8.GetFrequency

DirectSoundSecondaryBuffer8.SetFX

[#]Sets effects for sounds in the buffer. The buffer must not be playing.

object.**SetFX(** _
EffectsCount **As Long,** _
Buffers() **As DSEFFECTDESC,** _
lResultIDs() **As Long)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

EffectsCount
Number of elements in *Buffers* and *lResultIDs*. Set to 0 to remove all effects from the buffer.

Buffers
Array of **DSEFFECTDESC** types that specify the effects to set.

lResultIDs
Array of **Long** variables, one for each effect in the *Buffers* array, that receive information about the result of the attempt to create the corresponding effect. Each element receives one of the following values from the **CONST_DSOUNDFX** enumeration.

DSFXR_LOCHARDWARE
Effect is instantiated in hardware.

DSFXR_LOCSOFTWARE
Effect is instantiated in software.

DSFXR_UNALLOCATED
Effect has not yet been assigned to hardware nor software.

DSFXR_FAILED
No effect was created because resources weren't available.

DSFXR_PRESENT
The effect is available but was not created because the method failed for some other reason.

DSFXR_UNKNOWN
No effect was created because the effect isn't registered on the system.

[#] IDH_DirectSoundSecondaryBuffer8.SetFX_dxaudio_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_INVALIDCALL
DSERR_NOINTERFACE
DSERR_PRIOLEVELNEEDED

Remarks

For the method to succeed, the buffer must have been created with the DSBCAPS_CTRLFX flag and must not be playing.

If the method fails, the value for each effect in *lEffectsResults* is either DSFXF_PRESENT or DSFXR_UNKNOWN. Check these values to determine which effects caused the failure.

If the method returns DSERR_NOINTERFACE or another COM error, check the result code array for DSFXR_PRESENT or DSFXR_UNKNOWN to ascertain which effect caused the error. If the method returns DSERR_INVALIDPARAM, check the result codes for DSFXR_FAILED to see which effects failed to acquire resources.

If the buffer is part of an audiopath, the audiopath must be inactive. See **DirectMusicAudioPath8.Activate**.

An effect must be set on a buffer before the effect object can be obtained. To obtain the effect object, use **DirectSoundSecondaryBuffer8.GetObjectInPath**.

DirectSoundSecondaryBuffer8.SetNotificationPositions

#Sets the notification positions. During playback, whenever the play cursor reaches a specified offset, the associated event is signaled.

object.**SetNotificationPositions(** _
nElements **As Long**, _
psa() **As DSBPOSITIONNOTIFY)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

IDH_DirectSoundSecondaryBuffer8.SetNotificationPositions_dxaudio_vb

nElements
Number of elements in the *psa* array.

psa
Array of **DSBPOSITIONNOTIFY** types that specify the notification positions and associated events.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

See Also

Play Buffer Notification

DirectSoundSecondaryBuffer8.SetPan

[#]Specifies the relative volume between the left and right channels.

object.**SetPan(***pan* **As Long)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

pan
Value that specifies the relative attenuation of the channels. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

IDH_DirectSoundSecondaryBuffer8.SetPan_dxaudio_vb

Remarks

The value in *pan* is in hundredths of a decibel (dB), in the range of DSBPAN_LEFT (-10,000) to DSBPAN_RIGHT (10,000). The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER (0), meaning that both channels are at full volume (attenuated by 0 decibels).

At any setting other than DSBPAN_CENTER, one channel is at full volume and the other is attenuated. For example, a value 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.

The pan control acts cumulatively with the volume control.

See Also

DirectSoundSecondaryBuffer8.GetPan,
DirectSoundSecondaryBuffer8.SetVolume

DirectSoundSecondaryBuffer8.SetVolume

[#]Changes the volume of a sound buffer.

object.**SetVolume(***volume* **As Long)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

volume
Attenuation of the volume, in hundredths of a decibel. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

IDH_DirectSoundSecondaryBuffer8.SetVolume_dxaudio_vb

Remarks

The value of *volume* must be in the range from DSBVOLUME_MAX through DSBVOLUME_MIN. These constants are defined as 0 and -10,000 respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the sound. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for practical purposes, is silence.

See Also

DirectSoundSecondaryBuffer8.GetVolume,
DirectSoundSecondaryBuffer8.SetPan

DirectSoundSecondaryBuffer8.Stop

#Causes the sound buffer to stop playing.

object.**Stop()**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error values.

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The method sets the play cursor of the buffer to the sample that follows the last sample played. This means that when the **DirectSoundSecondaryBuffer8.Play** method is subsequently called on the buffer, it continues playing where it left off.

DirectSoundSecondaryBuffer8.WriteBuffer

#Writes to a segment of the sound buffer from an application buffer holding audio data.

IDH_DirectSoundSecondaryBuffer8.Stop_dxaudio_vb
IDH_DirectSoundSecondaryBuffer8.WriteBuffer_dxaudio_vb

object.**WriteBuffer(_**
start **As Long, _**
size **As Long, _**
buffer **As Any, _**
flags **As CONST_DSBLOCKFLAGS)**

Parts

object
Resolves to a **DirectSoundSecondaryBuffer8** object.

start
Offset, in bytes, from the start of the buffer to where the read begins. This parameter is ignored if DSBLOCK_FROMWRITECURSOR constant is specified in the *flags* parameter.

size
Size, in bytes, of the portion of the buffer to write to. The sound buffer is conceptually circular, so this value can be greater than the number of bytes between *start* and the end of the buffer. This parameter is ignored if DSBLOCK_ENTIREBUFFER constant is specified in the *flags* parameter.

buffer
Variable that contains the data to write to the sound buffer. Typically this is an array of bytes, and you pass the first element of the array.

flags
One of the constants of the **CONST_DSBLOCKFLAGS** enumeration specifying how to write the data.

Error Codes

If the method Fails, an error is raised and **Err.Number** may be set to one of the following error codes.

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

See Also

Using Streaming Buffers

DirectSound Types

This section contains reference information for the following types used with DirectSound.

- **DS3DBUFFER**
- **DS3DLISTENER**
- **DSBCAPS**
- **DSBPOSITIONNOTIFY**
- **DSBUFFERDESC**
- **DSCAPS**
- **DSCBCAPS**
- **DSCBUFFERDESC**
- **DSCCAPS**
- **DSCURSORS**
- **DSEFFECTDESC**
- **DSFXI3DL2REVERB**
- **DSFXCHORUS**
- **DSFXCOMPRESSOR**
- **DSFXDISTORTION**
- **DSFXECHO**
- **DSFXFLANGER**
- **DSFXGARGLE**
- **DSFXPARAMEQ**
- **DSFXWAVESREVERB**
- **WAVEFORMATEX**

DS3DBUFFER

#Describes the location, orientation, and motion of a 3-D sound buffer. This type is used with the **DirectSound3DBuffer8.GetAllParameters** and **DirectSound3DBuffer8.SetAllParameters** methods.

```
Type DS3DBUFFER
    lConeOutsideVolume As Long
    lInsideConeAngle As Long
    lMode As CONST_DS3DMODEFLAGS
    lOutsideConeAngle As Long
    maxDistance As Single
    minDistance As Single
```

IDH_DS3DBUFFER_dxaudio_vb

```
    vConeOrientation As D3DVECTOR
    vPosition As D3DVECTOR
    vVelocity As D3DVECTOR
  End Type
```

Members

lConeOutsideVolume
Volume outside the sound cone.

lInsideConeAngle
Angle of the inner sound projection cone.

lMode
3-D sound processing mode to set. This is one of the constants of the **CONST_DS3DMODEFLAGS** enumeration.

lOutsideConeAngle
Angle of the outer sound projection cone.

maxDistance
Maximum distance.

minDistance
Minimum distance.

vConeOrientation
D3DVECTOR type that describes the orientation of the sound projection cone.

vPosition
D3DVECTOR type that describes the position of the buffer.

vVelocity
D3DVECTOR type that describes the velocity of the buffer.

DS3DLISTENER

#Describes 3-D global parameters and properties of the listener. This type is used with the **DirectSound3DListener8.GetAllParameters** and **DirectSound3DListener8.SetAllParameters** methods.

```
  Type DS3DLISTENER
    distanceFactor As Single
    dopplerFactor As Single
    rolloffFactor As Single
    vOrientFront As D3DVECTOR
    vOrientTop As D3DVECTOR
    vPosition As D3DVECTOR
    vVelocity As D3DVECTOR
  End Type
```

IDH_DS3DLISTENER_dxaudio_vb

Members

distanceFactor
Number of meters in a unit of distance.

dopplerFactor
Factor applied to the Doppler shift for a buffer that has velocity.

rolloffFactor
Factor applied to the attenuation of sound over distance.

vOrientFront
D3DVECTOR type that describes the listener's front orientation.

vOrientTop
D3DVECTOR type that describes the listener's top orientation.

vPosition
D3DVECTOR type that describes the listener's position.

vVelocity
D3DVECTOR type that describes the listener's velocity.

DSBCAPS

#Describes the capabilities of a buffer object. It is used in the **DirectSoundPrimaryBuffer8.GetCaps** and **DirectSoundSecondaryBuffer8.GetCaps**method.

```
Type DSBCAPS
   lBufferBytes As Long
   lFlags As CONST_DSBCAPSFLAGS
   lPlayCpuOverhead As Long
   lUnlockTransferRate As Long
End Type
```

Members

lBufferBytes
Size of the buffer, in bytes.

lFlags
One of more of the constants of the **CONST_DSBCAPSFLAGS** enumeration.

lPlayCpuOverhead
Processing overhead, specified as a percentage of main processing cycles needed to mix this sound buffer. For hardware buffers, this value is zero, because the mixing is performed by the sound device. For software buffers, the value depends on the buffer format and the speed of the system processor.

lUnlockTransferRate
Rate, in kilobytes per second, at which data is transferred to the buffer memory.

IDH_DSBCAPS_dxaudio_vb

Remarks

The **DSBCAPS** type contains information similar to that found in the **DSBUFFERDESC** type passed to the **DirectSound8.CreateSoundBuffer** method.

The **lFlags** member of the **DSBCAPS** type contains the same flags used by the **DSBUFFERDESC** type. The only difference is that in the **DSBCAPS** type, either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag is returned, according to the location of the buffer memory. In the **DSBUFFERDESC** type, these flags are optional and are used to force the buffer to be located in either hardware or software.

DSBPOSITIONNOTIFY

#Describes a notification position. Used by the **DirectSoundSecondaryBuffer8.SetNotificationPositions** and **DirectSoundCaptureBuffer8.SetNotificationPositions** methods.

```
Type DSBPOSITIONNOTIFY
    hEventNotify As Long
    lOffset As Long
End Type
```

Members

hEventNotify
Handle to the event to signal when the offset has been reached. Obtain this handle by using the **DirectX8.CreateEvent** method.

lOffset
Offset from the beginning of the buffer where the notify event is to be triggered, or DSBPN_OFFSETSTOP. See Remarks.

Remarks

The DSBPN_OFFSETSTOP value in the **lOffset** member causes the event to be signaled when playback or capture stops, either because the end of the buffer has been reached and playback or capture is not looping, or because the application called the **DirectSoundSecondaryBuffer8.Stop** or **DirectSoundCaptureBuffer8.Stop** method.

When you use DSBCAPS_LOCDEFER and DSBCAPS_NOTIFY along with a voice management flag, a sound that has notifications set, but not yet reached, may be terminated by the voice manager. In this event, the notification event will not occur.

IDH_DSBPOSITIONNOTIFY_dxaudio_vb

DSBUFFERDESC

[#]Describes the necessary characteristics of a new DirectSound buffer object. This type is used by the **DirectSound8.CreateSoundBuffer** and **DirectSound8.CreatePrimarySoundBuffer** methods.

```
Type DSBUFFERDESC
   fxFormat As WaveFormatEx
   guid3DAlgorithm As String
   lBufferBytes As Long
   lFlags As CONST_DSBCAPSFLAGS
End Type
```

Members

fxFormat

WAVEFORMATEX type that specifies the wave format of the buffer.

guid3DAlgorithm

Unique identifier of the two-speaker virtualization algorithm to be used by DirectSound when 3-D effects are not available in hardware. If the DSCAPS_CTRL3D flag is not set for the sound buffer, this value is ignored. If DSBCAPS_CTRL3D is not set in **lFlags**, this member must be GUID_DS3DALG_DEFAULT. See Remarks.

The following algorithm identifiers are defined in the AUDIOSTRINGCONSTANTS module.

GUID_DS3DALG_DEFAULT

DirectSound uses the default algorithm. The default algorithm may be based on a Control Panel setting. In most cases it is DS3DALG_NO_VIRTUALIZATION. On WDM drivers, if the user has selected surround sound in Control Panel, the sound is panned among the left, center, right, and surround channels.

Applies to Software Mixing Only. Available on WDM or Vxd Drivers.

GUID_DS3DALG_NO_VIRTUALIZATION

3-D effects are mapped onto normal stereo panning. At 90 degrees to the left, the sound is coming out of only the left speaker; at 90 degrees to the right, sound is coming out of only the right speaker. The vertical axis is ignored except for scaling of volume due to distance. Doppler shift and volume scaling are still applied, but the 3-D filtering is not performed on this buffer. This is the most efficient software implementation, but provides no virtual 3-D audio effect. When the DS3DALG_NO_VIRTUALIZATION flag is specified, HRTF processing is not done. Since DS3DALG_NO_VIRTUALIZATION uses only normal stereo panning, a buffer created with this algorithm may be accelerated by a 2-D hardware voice if no free 3-D hardware voices are available.

[#] IDH_DSBUFFERDESC_dxaudio_vb

Applies to Software Mixing Only. Available on WDM or Vxd Drivers.

GUID_DS3DALG_HRTF_FULL
3-D effects are processed with the high quality 3-D audio algorithm. This algorithm gives the highest quality 3-D audio effect, but uses more CPU cycles. See Remarks.

Applies to Software Mixing Only. Available on Microsoft® Windows® 98 Second Edition and Windows 2000 when using WDM drivers.

GUID_DS3DALG_HRTF_LIGHT
3-D effects are processed with the efficient 3-D audio algorithm. This algorithm gives a good 3-D audio effect, but uses fewer CPU cycles than DS3DALG_HRTF_FULL. See Remarks.

Applies to Software Mixing Only. Available on Windows 98 Second Edition and Windows 2000 when using WDM drivers.

lBufferBytes
Size of the buffer, in bytes. This value must be 0 when creating a buffer with the DSBCAPS_PRIMARYBUFFER flag. For secondary buffers, the size must be in the range from 4 to &H0FFFFFFF (268,435,455).

lFlags
Flags specifying the capabilities of the buffer. Can be zero or one or more of the constants of the **CONST_DSBCAPSFLAGS** enumeration.

Remarks

The DSBCAPS_LOCHARDWARE and DSBCAPS_LOCSOFTWARE flags are optional and mutually exclusive. DSBCAPS_LOCHARDWARE forces the buffer to reside in hardware, meaning that it will be mixed by the sound card. DSBCAPS_LOCSOFTWARE forces the buffer to reside in software, where it is mixed by the CPU. These flags are also returned in the **lFlags** member of the **DSBCAPS** type, where they indicate the actual location of the buffer.

The 3-D algorithms represent selection of the software emulation layer only—that is, the software algorithm that is used when no hardware is present for acceleration. In order to maximize hardware utilization, DS3DALG_NO_VIRTUALIZATION is treated as a special case. If no free 3-D hardware voices are available, the buffer is then treated as a 2-D buffer, but with 3-D API control. Specifically, when a sound buffer is created with DS3DALG_NO_VIRTUALIZATION, or when the buffer is played if the buffer was created with DSBPLAY_LOCDEFER, the following procedure is followed:

- If a free hardware 3-D voice is available, that 3-D hardware voice is used.
- If no free hardware 3-D voices are available and a 2-D hardware voice is available, that 2-D hardware voice will be used. This is possible because the DS3DALG_NO_VIRTUALIZATION algorithm is a simple stereo pan algorithm
- If no free 2-D or 3-D hardware voices are available, the voice will be played in software using the DS3DALG_NO_VIRTUALIZATION algorithm.

If a speaker configuration other than DSSPEAKER_HEADPHONE or DSSPEAKER_STEREO is in effect, the processing will be done as if for a two-speaker configuration.

If a buffer is created using one of the HRTF algorithms, and the HRTF algorithm is not available on the system (for example, a non-WDM system), the method succeeds but the sound buffer uses DS3DALG_NO_VIRTUALIZATION instead.

DSCAPS

[#]Describes the capabilities of a DirectSound device. Used by the **DirectSound8.GetCaps** method.

```
Type DSCAPS
    lFlags As CONST_DSCAPSFLAGS
    lFreeHw3DAllBuffers As Long
    lFreeHw3DStaticBuffers As Long
    lFreeHw3DStreamingBuffers As Long
    lFreeHwMemBytes As Long
    lFreeHwMixingAllBuffers As Long
    lFreeHwMixingStaticBuffers As Long
    lFreeHwMixingStreamingBuffers As Long
    lMaxContigFreeHwMemBytes As Long
    lMaxHw3DAllBuffers As Long
    lMaxHw3DStaticBuffers As Long
    lMaxHw3DStreamingBuffers As Long
    lMaxHwMixingAllBuffers As Long
    lMaxHwMixingStaticBuffers As Long
    lMaxHwMixingStreamingBuffers As Long
    lMaxSecondarySampleRate As Long
    lMinSecondarySampleRate As Long
    lPlayCpuOverheadSwBuffers As Long
    lPrimaryBuffers As Long
    lReserved1 As Long
    lReserved2 As Long
    lTotalHwMemBytes As Long
    lUnlockTransferRateHwBuffers As Long
End Type
```

Members

lFlags

Device capabilities. Can be one or more of the constants from the **CONST_DSCAPSFLAGS** enumeration.

lFreeHw3DAllBuffers

[#] IDH_DSCAPS_dxaudio_vb

Number of unallocated 3-D buffers.

lFreeHw3DStaticBuffers
Number of unallocated static 3-D buffers.

lFreeHw3DStreamingBuffers
Number of unallocated streaming 3-D buffers.

lFreeHwMemBytes
Size, in bytes, of the free memory on the sound card.

lFreeHwMixingAllBuffers
Number of unallocated buffers. On WDM drivers, this includes **lFreeHw3DAllBuffers**.

lFreeHwMixingStaticBuffers
Number of unallocated static buffers.

lFreeHwMixingStreamingBuffers
Number of unallocated streaming buffers.

lMaxContigFreeHwMemBytes
Size, in bytes, of the largest contiguous block of free memory on the sound card.

lMaxHw3DAllBuffers
Maximum number of 3-D buffers.

lMaxHw3DStaticBuffers
Maximum number of static 3-D buffers.

lMaxHw3DStreamingBuffers
Maximum number of streaming 3-D buffers.

lMaxHwMixingAllBuffers
Total number of buffers that can be mixed in hardware. This member can be less than the sum of **lMaxHwMixingStaticBuffers** and **lMaxHwMixingStreamingBuffers**. Resource tradeoffs frequently occur.

lMaxHwMixingStaticBuffers
Maximum number of static sound buffers.

lMaxHwMixingStreamingBuffers
Maximum number of streaming sound buffers.

lMaxSecondarySampleRate, lMinSecondarySampleRate
Maximum and minimum sample rate specifications that are supported by this device's hardware secondary sound buffers.

lPlayCpuOverheadSwBuffers
Description of the processing overhead, as a percentage of the central processing unit, needed to mix software buffers—those located in main system memory. This varies according to the bus type, the processor type, and the clock speed.

The unlock transfer rate for software buffers is zero because the data need not be transferred. Similarly, the play processing overhead for hardware buffers is 0 because the mixing is done by the sound device.

lPrimaryBuffers
Number of primary buffers supported. This value is always 1.

lReserved1, lReserved2

Reserved.

lTotalHwMemBytes
Size, in bytes, of the memory on the sound card that stores static sound buffers.

lUnlockTransferRateHwBuffers
Rate, in kilobytes per second, at which data can be transferred to hardware static sound buffers.

Remarks

Some audio cards may be unable to report accurately the number of available or free hardware buffers. This can happen, for example, when the card can play more sounds at lower sampling rates than at higher rates. In general, a nonzero value in any of the members relating to number of free hardware buffers signifies that at least one hardware resource of the appropriate type is available.

DSCBCAPS

#Describes the capabilities of a capture buffer. Used by the **DirectSoundCaptureBuffer8.GetCaps** method.

```
Type DSCBCAPS
    lBufferBytes As Long
    lFlags As CONST_DSCBCAPSFLAGS
    lReserved As Long
End Type
```

Members

lBufferBytes
Size, in bytes, of the capture buffer.

lFlags
Flag describing device capabilities. One of the constants of the **CONST_DSCBCAPSFLAGS** enumeration.

lReserved
Reserved.

DSCBUFFERDESC

#Describes a capture buffer. Used by the **DirectSoundCapture8.CreateCaptureBuffer** method.

```
Type DSCBUFFERDESC
    fxFormat As WAVEFORMATEX
```

IDH_DSCBCAPS_dxaudio_vb
IDH_DSCBUFFERDESC_dxaudio_vb

```
    guid3DAlgorithm As String
    lBufferBytes As Long
    lFlags As CONST_DSCBCAPSFLAGS
End Type
```

Members

fxFormat
WAVEFORMATEX type containing the format in which to capture the data.

guid3DAlgorithm
Not used.

lBufferBytes
Size of capture buffer to create, in bytes.

lFlags
Flag specifying device capabilities. One of the constants of the **CONST_DSCBCAPSFLAGS** enumeration.

DSCCAPS

[#]Describes the capabilities of the capture device. Used by the **DirectSoundCapture8.GetCaps** method.

```
Type DSCCAPS
    lChannels As Long
    lFlags As CONST_DSCCAPSFLAGS
    lFormats As CONST_WAVEFORMATFLAGS
End Type
```

Members

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

lFlags
Device capabilities. One of the members of the **CONST_DSCCAPSFLAGS** enumeration.

lFormats
Standard formats that are supported. One or more of the constants of the **CONST_WAVEFORMATFLAGS** enumeration.

IDH_DSCCAPS_dxaudio_vb

DSCURSORS

#Returns the position of the play and write cursors to the **DirectSoundSecondaryBuffer8.GetCurrentPosition** method. Also returns the position of the capture and read cursors to the **DirectSoundCaptureBuffer8.GetCurrentPosition** method.

```
Type DSCURSORS
   lPlay As Long
   lWrite As Long
End Type
```

Members

lPlay
Play cursor in the play buffer or capture cursor in the capture buffer.

lWrite
Write cursor in the play buffer or read cursor in the capture buffer.

Remarks

Values are offsets from the start of the buffer, in bytes.

The cursor represented by **lPlay** is slightly ahead of the cursor represented by **lWrite**. The part of the buffer between the two is not ready for reading or writing. Applications can safely read data from a capture buffer or write data to a play buffer up to **lWrite**.

See Also

Play and Write Cursors, Capture Buffer Information

DSEFFECTDESC

#Describes an effect associated with a buffer.

```
Type DSEFFECTDESC
   guidDSFXClass As String
   lFlags As Long
End Type
```

Members

guidDSFXClass

IDH_DSCURSORS_dxaudio_vb
IDH_DSEFFECTDESC_dxaudio_vb

String that specifies the class identifier of the effect. You can use one of the following aliases for a standard effect, or a GUID.

DSFX_STANDARD_CHORUS
DSFX_STANDARD_COMPRESSOR
DSFX_STANDARD_DISTORTION
DSFX_STANDARD_ECHO
DSFX_STANDARD_FLANGER
DSFX_STANDARD_GARGLE
DSFX_STANDARD_I3DL2REVERB
DSFX_STANDARD_PARAMEQ
DSFX_STANDARD_WAVES_REVERB

lFlags

Flags that specify buffer creation options. Can be 0 or one of the following values from the **CONST_DSOUNDFX** enumeration.

DSFX_LOCHARDWARE
Effect must be in hardware. If the effect is not available in hardware, **DirectSoundSecondaryBuffer8.SetFX** raises an error. Because DirectX 8.0 does not support hardware acceleration of effects, this flag should not be used.

DSFX_LOCSOFTWARE
Effect must be in software, even if the hardware supports acceleration of **guidDSFXClass**. If the effect is not available in software, **SetFX** raises an error. In DirectX 8.0, all effects are in software regardless of whether this flag is set.

Remarks

If **lFlags** is zero, the effect is placed in hardware if possible. If the hardware does not support the effect, software is used. If the effect is not available at all, the call to **SetFX** raises an error.

An effect of class DSFX_STANDARD_WAVES_REVERB can be set only on a buffer that has a 16-bit audio format.

See Also

Using Effects

DSFXI3DL2REVERB

[#]Contains parameters for an I3DL2 (Interactive 3D Audio Level 2) reverberation effect.

```
Type DSFXI3DL2REVERB
    fDecayHFRatio As Single
```

IDH_DSFX_I3DL2REVERB_dxaudio_vb

```
    fDecayTime As Single
    fDensity As Single
    fDiffusion As Single
    fHFReference As Single
    fReflectionsDelay As Single
    fReverbDelay As Single
    fRoomRolloffFactor As Single
    lReflections As Long
    lReverb As Long
    lRoom As Long
    lRoomHF As Long
End Type
```

Members

fDecayHFRatio
Ratio of the decay time at high frequencies to the decay time at low frequencies, in the range 0.1 to 2. The default value is 0.83.

fDecayTime
Decay time, in the range 0.1 to 20 seconds. The default value is 1.49 seconds.

fDensity
Modal density in the late reverberation decay. The default value is 100.0 percent. Minimum and maximum values are defined in the **CONST_DSFX_I3DL2REVERB_DENSITY** enumeration.

fDiffusion
Echo density in the late reverberation decay. The default value is 100.0 percent. Minimum and maximum values are defined in the **CONST_DSFX_I3DL2REVERB_DIFFUSION** enumeration.

fHFReference
Reference high frequency. The default value is 5000.0 Hz. Minimum and maximum values are defined in the **CONST_DSFX_I3DL2REVERB_HFREFERENCE** enumeration.

fReflectionsDelay
Delay time of the first reflection relative to the direct path, in the range 0 to 0.3 seconds. The default value is 0.007 seconds.

fReverbDelay
Time limit between the early reflections and the late reverberation relative to the time of the first reflection, in the range 0 to 0.1 seconds. The default value is 0.011 seconds.

fRoomRolloffFactor
Rolloff factor for the reflected signals. The default value is 0.0. Minimum and maximum values are defined in the **CONST_DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR** enumeration. The rolloff factor for the direct path is controlled by the DirectSound listener.

lReflections
Attenuation of early reflections relative to **lRoom**. The default value is -2602 mB. Minimum and maximum values are defined in the **CONST_DSFX_I3DL2REVERB_REFLECTIONS** enumeration.

lReverb
Attenuation of late reverberation relative to **lRoom**. The default value is -200 mB. Minimum and maximum values are defined in the **CONST_DSFX_I3DL2REVERB_REVERB** enumeration.

lRoom
Attenuation of the room effect. The default value is -1000 mB. Minimum and maximum values are defined in the **CONST_DSFX_I3DL2REVERB_ROOM** enumeration.

lRoomHF
Attenuation of the room high-frequency effect. The default value is 0 mB. Minimum and maximum values are defined in the **CONST_DSFX_I3DL2REVERB_ROOMHF** enumeration.

DSFXCHORUS

[#]Contains parameters for a chorus effect.

```
Type DSFXCHORUS
   fDelay As Single
   fDepth As Single
   fFeedback As Single
   fFrequency As Single
   fWetDryMix As Single
   lPhase As CONST_DSFX_PHASE
   lWaveform As CONST_DSFX_WAVE
End Type
```

Members

fDelay
Amount of time, in milliseconds, that the input is delayed before it is played back. The default value is 0. Maximum and minimum values are defined in the **CONST_DSFX_DELAY** enumeration.

fDepth
Percentage by which the delay time is modulated by the low-frequency oscillator, in percentage points. The default value is 25. Maximum and minimum values are defined in the **CONST_DSFX_DEPTH** enumeration.

fFeedback

IDH_DSFXCHORUS_dxaudio_vb

Percentage of output signal to feed back into the effect's input. The default value is 0. Maximum and minimum values are defined in the **CONST_DSFX_FEEDBACK** enumeration.

fFrequency
Frequency of the LFO. The default value is 0. Maximum and minimum values are defined in the **CONST_DSFX_FREQUENCY** enumeration.

fWetDryMix
Ratio of wet (processed) signal to dry (unprocessed) signal. Minimum and maximum values are defined in the **CONST_DSFX_WETDRY** enumeration.

lPhase
Phase differential between left and right LFOs. Allowable values are defined in the **CONST_DSFX_PHASE** enumeration. The default value is 0.

lWaveform
Waveform of the LFO. By default, the waveform is a sine. Allowable values are defined in the **CONST_DSFX_WAVE** enumeration.

Remarks

Where the type of a member is an enumeration that defines minimum and maximum values, other **Long** values within this defined range are allowed. The enumeration is a convenience so that minimum and maximum values are easily accessible when writing code.

See Also

DirectSoundFXChorus8, Chorus

DSFXCOMPRESSOR

[#]Contains parameters for a compression effect.

```
Type DSFXCOMPRESSOR
   fAttack As Single
   fGain As Single
   fPreDelay As Single
   fRatio As Single
   fRelease As Single
   fThreshold As Single
End Type
```

Members

fAttack

IDH_DSFXCOMPRESSOR_dxaudio_vb

Time before compression reaches its full value. The default value is 0.01 ms. Minimum and maximum values are defined in the **CONST_DSFXCOMPRESSOR_ATTACK** enumeration. See Remarks.

fGain

Output gain of signal after compression. The default value is 0 dB. Minimum and maximum values are defined in the **CONST_DSFXCOMPRESSOR_GAIN** enumeration.

fPreDelay

Time after **lThreshold** is reached before attack phase is started. The default value is 0 ms. Minimum and maximum values are defined in the **CONST_DSFXCOMPRESSOR_PREDELAY** enumeration.

fRatio

Compression ratio. The default value is 10, which means 10:1 compression. Minimum and maximum values are defined in the **CONST_DSFXCOMPRESSOR_RATIO** enumeration.

fRelease

Speed at which compression is stopped after input drops below **lThreshold**. The default value is 50 ms. Minimum and maximum values are defined in the **CONST_DSFXCOMPRESSOR_RELEASE** enumeration.

fThreshold

Point at which compression begins. The default value is -10 dB. Minimum and maximum values are defined in the **CONST_DSFXCOMPRESSOR_THRESHOLD** enumeration.

Remarks

For convenience, the minimum value for **fAttack** is defined as the integer 1 in the **CONST_DSFXCOMPRESSOR_ATTACK** enumeration. However, the actual minimum value, which is also the default, is 0.01.

See Also

DirectSoundFXCompressor8, Compression

DSFXDISTORTION

#Contains parameters for a distortion effect.

```
Type DSFXDISTORTION
    fEdge As Single
    fGain As Single
    fPostEQBandwidth As Single
    fPostEQCenterFrequency As Single
    fPreLowpassCutoff As Single
```

IDH_DSFXDISTORTION_dxaudio_vb

```
End Type
```

Members

fEdge
Percentage of distortion intensity. The default value is 50. Minimum and maximum values are defined in the **CONST_DSFXDISTORTION_EDGE** enumeration.

fGain
Amount of signal change after distortion, in decibels. The default value is 0. Minimum and maximum values are defined in the **CONST_DSFXDISTORTION_GAIN** enumeration.

fPostEQBandwidth
Width of frequency band that determines range of harmonic content addition, in Hertz. The default value is 4000. Minimum and maximum values are defined in the **CONST_DSFXDISTORTION_POSTEQBANDWIDTH** enumeration.

fPostEQCenterFrequency
Center frequency of harmonic content addition, in Hertz. The default value is 4000. Minimum and maximum values are defined in the **CONST_DSFXDISTORTION_POSTEQCENTERFREQUENCY** enumeration.

fPreLowpassCutoff
Filter cutoff for high-frequency harmonics attenuation, in Hertz. The default value is 4000. Minimum and maximum values are defined in the **CONST_DSFXDISTORTION_PRELOWPASSCUTOFF** enumeration.

Remarks

The values in **fPostEQBandwidth**, **fPostEQCenterFrequency**, and **fPreLowpassCutoff** cannot exceed one-third of the frequency of the buffer.

See Also

DirectSoundFXDistortion8, Distortion

DSFXECHO

#Contains parameters for an echo effect.

```
Type DSFXECHO
    fFeedback As Single
    fLeftDelay As Single
    fRightDelay As Single
    fWetDryMix as Single
```

IDH_DSFXECHO_dxaudio_vb

```
    lPanDelay As CONST_DSFX_PANDELAY
End Type
```

Members

fFeedback
Percentage of output fed back into input. The default value is 0. Minimum and maximum values are defined in the **CONST_DSFXECHO_FEEDBACK** enumeration.

fLeftDelay
Delay for left channel. The default value is 333 ms. Minimum and maximum values are defined in the **CONST_DSFX_LEFTRIGHTDELAY** enumeration.

fRightDelay
Delay for right channel. The default value is 333 ms. Minimum and maximum values are defined in the **CONST_DSFX_LEFTRIGHTDELAY** enumeration.

fWetDryMix
Ratio of wet (processed) signal to dry (unprocessed) signal. Minimum and maximum values are defined in the **CONST_DSFX_WETDRY** enumeration.

lPanDelay
Value that specifies whether to swap left and right delays with each successive echo. The default value is zero, meaning no swap. Allowable values are defined in the **CONST_DSFX_PANDELAY** enumeration.

Remarks

Where the type of a member is an enumeration that defines minimum and maximum values, other **Long** values within this defined range are allowed. The enumeration is a convenience so that minimum and maximum values are easily accessible when writing code.

See Also

DirectSoundFXEcho8, Echo

DSFXFLANGER

#Contains parameters for a flanging effect.

```
Type DSFXFLANGER
    fDelay As Single
    fDepth As Single
    fFeedback As Single
    fFrequency As Single
    fWetDryMix As Single
```

IDH_DSFXFLANGER_dxaudio_vb

```
    lPhase As CONST_DSFX_PHASE
    lWaveform As CONST_DSFX_WAVE
End Type
```

Members

fDelay
Amount of time, in milliseconds, that the input is delayed before it is played back. The default value is 0. Maximum and minimum values are defined in the **CONST_DSFX_FLANGER_DELAY** enumeration.

fDepth
Percentage by which the delay time is modulated by the low-frequency oscillator, in percentage points. The default value is 25. Maximum and minimum values are defined in the **CONST_DSFX_DEPTH** enumeration.

fFeedback
Percentage of output signal to feed back into the effect's input. The default value is 0. Maximum and minimum values are defined in the **CONST_DSFX_FEEDBACK** enumeration.

fFrequency
Frequency of the LFO. The default value is 0. Maximum and minimum values are defined in the **CONST_DSFX_FREQUENCY** enumeration.

fWetDryMix
Ratio of wet (processed) signal to dry (unprocessed) signal. Minimum and maximum values are defined in the **CONST_DSFX_WETDRY** enumeration.

lPhase
Phase differential between left and right LFOs. Allowable values are defined in the **CONST_DSFX_PHASE** enumeration. The default value is 0.

lWaveform
Waveform of the LFO. By default, the waveform is a sine. Allowable values are defined in the **CONST_DSFX_WAVE** enumeration.

Remarks

Where the type of a member is an enumeration that defines minimum and maximum values, other **Long** values within this defined range are allowed. The enumeration is a convenience so that minimum and maximum values are easily accessible when writing code.

See Also

DirectSoundFXFlanger8, Flange

DSFXGARGLE

#Contains parameters for an amplitude modulation effect represented by a **DirectSoundFXGargle8** object.

```
Type DSFXGARGLE
   lRateHz As CONST_DSFXGARGLE_RATEHZ
   lWaveShape As CONST_DSFXGARGLE_WAVE
End Type
```

Members

lRateHz
Rate of modulation, in Hertz. Minimum and maximum values are defined in the **CONST_DSFXGARGLE_RATEHZ** enumeration. Any **Long** value within this range is valid.

lWaveShape
Shape of the modulation wave. Allowable values are defined in the **CONST_DSFXGARGLE_WAVE** enumeration.

See Also

DirectSoundFXGargle8, Gargle

DSFXPARAMEQ

#Contains parameters for a parametric equalizer effect.

```
Type DSFXPARAMEQ
   fBandwidth As Single
   fCenter As Single
   fGain As Single
End Type
```

Members

fBandwidth
Bandwidth, in semitones. Minimum and maximum values are defined in the **CONST_DSFXPARAMEQ_BANDWIDTH** enumeration.

fCenter
Center frequency, in hertz. Minimum and maximum values are defined in the **CONST_DSFXPARAMEQ_CENTER** enumeration. This value cannot exceed one-third of the frequency of the buffer.

IDH_DSFXGARGLE_dxaudio_vb
IDH_DSFXPARAMEQ_dxaudio_vb

fGain
Gain, in decibels. Minimum and maximum values are defined in the **CONST_DSFXPARAMEQ_GAIN** enumeration.

See Also

DirectSoundFXParamEq8, Parametric Equalizer

DSFXWAVESREVERB

#Contains parameters for a music reverberation effect represented by a **DirectSoundFXWavesReverb8** object.

```
Type DSFXWavesReverb
   fHighFreqRTRatio As Long
   fInGain As Long
   fReverbMix As Long
   fReverbTime As Long
End Type
```

Members

fHighFreqRTRatio
In the range from 0.001 through 0.999. The default value is 0.001.

fInGain
Input gain of signal, in decibels (dB). The default value is 0.0. Maximum and minimum values are defined in the **CONST_DSFX_WAVESREVERB_INGAIN** enumeration.

fReverbMix
Reverb mix, in dB. The default value is 0. Maximum and minimum values are defined in the **CONST_DSFX_WAVESREVERB_REVERBMIX** enumeration.

fReverbTime
Reverb time, in milliseconds, in the range from 0.001 through 3000.0. The default value is 1000.0.

WAVEFORMATEX

#Defines the format of waveform audio data.

```
Type WAVEFORMATEX
   lAvgBytesPerSec As Long
   lExtra As Long
```

IDH_DSFX_WAVESREVERB_dxaudio_vb
IDH_WAVEFORMATEX_dxaudio_vb

```
    lSamplesPerSec As Long
    nBitsPerSample As Integer
    nBlockAlign As Integer
    nChannels As Integer
    nFormatTag As Integer
    nSize As Integer
End Type
```

Members

lAvgBytesPerSec
Required average data-transfer rate, in bytes per second, for the format. If **nFormatTag** is WAVE_FORMAT_PCM, **nAvgBytesPerSec** should be equal to the product of **lSamplesPerSec** and **nBlockAlign**.

lExtra
Not used; set to zero.

lSamplesPerSec
Sample rate, in samples per second (hertz), at which each channel should be played or recorded. If **nFormatTag** is WAVE_FORMAT_PCM, then common values for **lSamplesPerSec** are 8.0 kHz, 11.025 kHz, 22.05 kHz, and 44.1 kHz.

nBitsPerSample
Bits per sample. If **nFormatTag** is WAVE_FORMAT_PCM, **nBitsPerSample** should be 8 or 16.

nBlockAlign
Block alignment, in bytes. The block alignment is the minimum atomic unit of data for the **nFormatTag** format type. If **nFormatTag** is WAVE_FORMAT_PCM, **nBlockAlign** should be equal to the product of **nChannels** and **nBitsPerSample** divided by 8 (bits per byte).

Data written to and read from a buffer must always start at the beginning of a block. For example, it is illegal to start playback of PCM data in the middle of a sample—that is, on a non-block-aligned boundary.

nChannels
Number of channels in the waveform-audio data. Monaural data uses one channel and stereo data uses two channels.

nFormatTag
Waveform-audio format type. In DirectSound, this member must always be WAVE_FORMAT_PCM.

nSize
Size, in bytes, of extra format information. For WAVE_FORMAT_PCM formats, this member is ignored.

DirectSound Enumerations

DirectSound uses enumerations to group constants to take advantage of the statement completion feature of Visual Basic. The enumerations used in DirectSound are:

- **CONST_DS3DAPPLYFLAGS**
- **CONST_DS3DMODEFLAGS**
- **CONST_DSBCAPSFLAGS**
- **CONST_DSBLOCKFLAGS**
- **CONST_DSBPLAYFLAGS**
- **CONST_DSBSTATUSFLAGS**
- **CONST_DSCAPSFLAGS**
- **CONST_DSCBCAPSFLAGS**
- **CONST_DSCBLOCKFLAGS**
- **CONST_DSCBSTARTFLAGS**
- **CONST_DSCBSTATUSFLAGS**
- **CONST_DSCCAPSFLAGS**
- **CONST_DSFX_DELAY**
- **CONST_DSFX_DEPTH**
- **CONST_DSFXECHO_FEEDBACK**
- **CONST_DSFX_FEEDBACK**
- **CONST_DSFX_FLANGER_DELAY**
- **CONST_DSFX_FREQUENCY**
- **CONST_DSFX_I3DL2_ENVIRONMENT_PRESETS**
- **CONST_DSFX_I3DL2REVERB_DENSITY**
- **CONST_DSFX_I3DL2REVERB_DIFFUSION**
- **CONST_DSFX_I3DL2REVERB_HFREFERENCE**
- **CONST_DSFX_I3DL2REVERB_QUALITY**
- **CONST_DSFX_I3DL2REVERB_REFLECTIONS**
- **CONST_DSFX_I3DL2REVERB_REVERB**
- **CONST_DSFX_I3DL2REVERB_ROOM•**
  CONST_DSFX_I3DL2REVERB_ROOMHF
- **CONST_DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR**
- **CONST_DSFX_LEFTRIGHTDELAY**
- **CONST_DSFX_PANDELAY**
- **CONST_DSFX_PHASE**
- **CONST_DSFX_WAVE**
- **CONST_DSFX_WAVESREVERB_INGAIN**

- **CONST_DSFX_WAVESREVERB_REVERBMIX**
- **CONST_DSFX_WETDRY**
- **CONST_DSFXCOMPRESSOR_ATTACK**
- **CONST_DSFXCOMPRESSOR_GAIN**
- **CONST_DSFXCOMPRESSOR_PREDELAY**
- **CONST_DSFXCOMPRESSOR_RATIO**
- **CONST_DSFXCOMPRESSOR_RELEASE**
- **CONST_DSFXCOMPRESSOR_THRESHOLD**
- **CONST_DSFXDISTORTION_EDGE**
- **CONST_DSFXDISTORTION_GAIN**
- **CONST_DSFXDISTORTION_POSTEQBANDWIDTH**
- **CONST_DSFXDISTORTION_POSTEQCENTERFREQUENCY**
- **CONST_DSFXDISTORTION_PRELOWPASSCUTOFF**
- **CONST_DSFXGARGLE_RATEHZ**
- **CONST_DSFXGARGLE_WAVE**
- **CONST_DSFXPARAMEQ_BANDWIDTH**
- **CONST_DSFXPARAMEQ_CENTER**
- **CONST_DSFXPARAMEQ_GAIN**• **CONST_DSOUND**
- **CONST_DSOUNDERR**
- **CONST_DSOUNDFX**
- **CONST_DSSCLFLAGS**
- **CONST_DSSPEAKERFLAGS**
- **CONST_WAVEFORMATFLAGS**

CONST_DS3DAPPLYFLAGS

[#]Used in parameters to various methods of the **DirectSound3DBuffer8** and **DirectSound3DListener8** classes. The members of this enumeration are used to specify when new settings should be applied.

```
Enum CONST_DS3DAPPLYFLAGS
   DS3D_DEFERRED  = 1
   DS3D_IMMEDIATE = 0
End Enum
```

Constants

DS3D_DEFERRED

[#] IDH_CONST_DS3DAPPLYFLAGS_dxaudio_vb

Settings are not applied until the application calls the **DirectSound3DListener8.CommitDeferredSettings** method.

DS3D_IMMEDIATE
Settings are applied immediately.

Remarks

When several parameter changes are made at once, it is more efficient to flag them as DS3D_DEFERRED and commit them with a single call.

CONST_DS3DMODEFLAGS

#Used by the **DirectSound3DBuffer8.GetMode** and **DirectSound3DBuffer8.SetMode** methods to set and retrieve the operation mode for 3-D sound processing.

```
Enum CONST_DS3DMODEFLAGS
   DS3DMODE_DISABLE      = 2
   DS3DMODE_HEADRELATIVE = 1
   DS3DMODE_NORMAL       = 0
End Enum
```

Constants

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

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

DS3DMODE_NORMAL
Sound parameters are relative to world space. This is the default mode.

CONST_DSBCAPSFLAGS

#Used in the **DSBUFFERDESC** types to specify desired capabilities of a new buffer and in **DSBCAPS** to describe capabilities of an existing buffer.

```
Enum CONST_DSBCAPSFLAGS
   DSBCAPS_CTRL3D            =     16 (&H10)
   DSBCAPS_CTRLCHANNELVOLUME   =   1024 (&H400)
   DSBCAPS_CTRLFREQUENCY       =     32 (&H20)
```

IDH_CONST_DS3DMODEFLAGS_dxaudio_vb
IDH_CONST_DSBCAPSFLAGS_dxaudio_vb

```
    DSBCAPS_CTRLFX           =    512 (&H200)
    DSBCAPS_CTRLPAN          =     64 (&H40)
    DSBCAPS_CTRLPOSITIONNOTIFY  =    256
    DSBCAPS_CTRLVOLUME       =    128 (&H80)
    DSBCAPS_GETCURRENTPOSITION2 =  65536 (&H10000)
    DSBCAPS_GLOBALFOCUS      =  32768 (&H8000)
    DSBCAPS_LOCDEFER         = 262144 (&H40000)
    DSBCAPS_LOCHARDWARE      =      4
    DSBCAPS_LOCSOFTWARE      =      8
    DSBCAPS_MUTE3DATMAXDISTANCE = 131072 (&H20000)
    DSBCAPS_PRIMARYBUFFER    =      1
    DSBCAPS_STATIC           =      2
    DSBCAPS_STICKYFOCUS      =  16384 (&H4000)
End Enum
```

Constants

DSBCAPS_CTRL3D
The buffer supports 3-D control.

DSBCAPS_CTRLCHANNELVOLUME
Not implemented.

DSBCAPS_CTRLFREQUENCY
The buffer supports frequency control. Cannot be combined with DSBCAPS_CTRLFX.

DSBCAPS_CTRLFX
The buffer supports effects. Cannot be combined with DSBCAPS_CTRLFREQUENCY. The wave format must be an 8-bit or 16-bit PCM format with no more than two channels.

DSBCAPS_CTRLPAN
The buffer supports pan control.

DSBCAPS_CTRLPOSITIONNOTIFY
The buffer supports position notification. See Remarks.

DSBCAPS_CTRLVOLUME
The buffer supports volume control.

DSBCAPS_GETCURRENTPOSITION2
DirectSoundSecondaryBuffer8.GetCurrentPosition uses the new behavior of the play cursor. Always set this flag on secondary buffers.

DSBCAPS_GLOBALFOCUS
The buffer is a global sound buffer. With this flag set, an application using DirectSound can continue to play its buffers if the user switches focus to another application, even if the new application uses DirectSound. However, if the focus is switched to a DirectSound application that uses the DSSCL_WRITEPRIMARY flag for its cooperative level, the global sounds from other applications are not audible.

DSBCAPS_LOCDEFER
The buffer is not assigned to hardware or software until it is played. This flag must be set for buffers that use voice management.

DSBCAPS_LOCHARDWARE
The buffer uses hardware mixing. If the device does not support hardware mixing or if the required hardware memory is not available, the call to the **DirectSound8.CreateSoundBuffer** method fails. The application must ensure that a mixing channel will be available for this buffer; this condition is not guaranteed.

DSBCAPS_LOCSOFTWARE
The buffer uses software mixing, even if DSBCAPS_STATIC is specified and hardware resources are available.

DSBCAPS_MUTE3DATMAXDISTANCE
The buffer stops playing when the maximum distance from the listener is exceeded, so that processor time is not wasted.

DSBCAPS_PRIMARYBUFFER
The buffer is a primary sound buffer. Cannot be combined with DSBCAPS_CTRLFX

DSBCAPS_STATIC
The buffer is placed in memory on the sound card, if available. This is not the same as DSBCAPS_LOCHARDWARE, which forces the buffer to be placed in memory, usually system memory, managed by the driver. Cannot be combined with DSBCAPS_CTRLFX.

DSBCAPS_STICKYFOCUS
The buffer has sticky focus. An application using DirectSound can continue to play such buffers if the user switches to an application that is not using DirectSound. In this situation, the application's normal buffers are muted, but the sticky focus buffers are still audible. This is useful for nongame applications when the user wants to hear the soundtrack while using another application. However, if the user switches to a DirectSound application, all sound buffers, both normal and sticky focus, in the previous application are muted.

Remarks

When a buffer is being created, the DSBCAPS_LOCHARDWARE and DSBCAPS_LOCSOFTWARE flags are optional and mutually exclusive. DSBCAPS_LOCHARDWARE forces the buffer to reside in hardware, meaning that it will be mixed by the sound card. DSBCAPS_LOCSOFTWARE forces the buffer to reside in software, where it is mixed by the CPU. In the **DSBCAPS** type, the flag indicates the actual location of the buffer.

If you create a sound buffer with the DSBCAPS_CTRLPOSITIONNOTIFYflag but fail to set notifications, the behavior is undefined, and sounds may be played twice.

On VxD drivers, if a sound buffer is created with DSBCAPS_CTRLPOSITIONNOTIFY, the buffer is always a software buffer.

Calling **DirectSoundSecondaryBuffer8.Play** with the DSBPLAY_LOCHARDWARE flag will fail when playing a buffer created with this flag set. With WDM drivers, a notification-enabled buffer can be in hardware, if hardware is available.

CONST_DSBLOCKFLAGS

#Used in the *flags* parameter of the **DirectSoundSecondaryBuffer8.ReadBuffer** and **DirectSoundSecondaryBuffer8.WriteBuffer** methods to specify what portion of the buffer is to be read or written to.

```
Enum CONST_DSBLOCKFLAGS
   DSBLOCK_DEFAULT        = 0
   DSBLOCK_ENTIREBUFFER    = 2
   DSBLOCK_FROMWRITECURSOR = 1
End Enum
```

Constants

DSBLOCK_DEFAULT
: Read from or write to the position specified in the *start* parameter.

DSBLOCK_ENTIREBUFFER
: Read from or write to the entire buffer. The *size* parameter is ignored.

DSBLOCK_FROMWRITECURSOR
: Read from or write to the write cursor. The *start* parameter is ignored.

CONST_DSBPLAYFLAGS

#Used in the *flags* parameter of the **DirectSoundSecondaryBuffer8.Play** method to specify how to play the buffer.

```
Enum CONST_DSBPLAYFLAGS
   DSBPLAY_DEFAULT           = 0
   DSBPLAY_LOCHARDWARE        = 2
   DSBPLAY_LOCSOFTWARE        = 4
   DSBPLAY_LOOPING           = 1
   DSBPLAY_TERMINATEBY_DISTANCE = 16 (&H10)
   DSBPLAY_TERMINATEBY_PRIORITY = 32 (&H20)
   DSBPLAY_TERMINATEBY_TIME    = 8
End Enum
```

IDH_CONST_DSBLOCKFLAGS_dxaudio_vb
IDH_CONST_DSBPLAYFLAGS_dxaudio_vb

Constants

DSBPLAY_DEFAULT
Play from the play cursor to the end of the audio buffer.

DSBPLAY_LOCHARDWARE
Play this voice in a hardware buffer only. If the hardware has no available voices and no voice management flags are set, the call to **DirectSoundsSecondaryBuffer8.Play** fails. This flag cannot be combined with DSBPLAY_LOCSOFTWARE.

DSBPLAY_LOCSOFTWARE
Play this voice in a software buffer only. This flag cannot be combined with DSBPLAY_LOCHARDWARE or any DSBPLAY_TERMINATEBY flag.

DSBPLAY_LOOPING
Play from the play cursor to the end of the buffer, and restart at the beginning of the buffer. The buffer plays till explicitly stopped.

DSBPLAY_TERMINATEBY_DISTANCE
If the hardware has no available voices, a currently playing buffer will be stopped to make room for the new buffer. The buffer prematurely terminated will be selected from buffers that have the buffer's DSBCAPS_ MUTE3DATMAXDISTANCE flag set and are beyond their maximum distance. If there are no such buffers, the method fails.

DSBPLAY_TERMINATEBY_PRIORITY
If the hardware has no available voices, a currently playing buffer will be stopped to make room for the new buffer.

DSBPLAY_TERMINATEBY_TIME
If the hardware has no available voices, a currently playing buffer will be stopped to make room for the new buffer. The buffer prematurely terminated is the one with the least time left to play.

CONST_DSBSTATUSFLAGS

[#]Describe the status of a buffer. Constants from this enumeration are returned by the **DirectSoundSecondaryBuffer8.GetStatus** method.

```
Enum CONST_DSBSTATUSFLAGS
   DSBSTATUS_BUFFERLOST  =  2
   DSBSTATUS_LOCHARDWARE =  8
   DSBSTATUS_LOCSOFTWARE = 16 (&H10)
   DSBSTATUS_LOOPING     =  4
   DSBSTATUS_PLAYING     =  1
   DSBSTATUS_TERMINATED  = 32 (&H20)
End Enum
```

IDH_CONST_DSBSTATUSFLAGS_dxaudio_vb

Constants

DSBSTATUS_BUFFERLOST
The buffer is lost and must be restored before it can be played or locked.

DSBSTATUS_LOCHARDWARE
The buffer is playing in hardware. Set only for buffers created with the DSBCAPS_LOCDEFER flag.

DSBSTATUS_LOCSOFTWARE
The buffer is playing in software. Set only for buffers created with the DSBCAPS_LOCDEFER flag.

DSBSTATUS_LOOPING
The buffer is being looped. If this value is not set, the buffer stops when it reaches the end of the sound data. This value is returned only in combination with DSBSTATUS_PLAYING.

DSBSTATUS_PLAYING
The buffer is playing.

DSBSTATUS_TERMINATED
The buffer was prematurely terminated by the voice manager and is not playing. Set only for buffers created with the DSBCAPS_LOCDEFER flag.

CONST_DSCAPSFLAGS

[#]Used in the **lFlags** member of the **DSCAPS** type to describe device capabilities.

```
Enum CONST_DSCAPSFLAGS
   DSCAPS_CERTIFIED      =   64 (&H40)
   DSCAPS_CONTINUOUSRATE =   16 (&H10)
   DSCAPS_EMULDRIVER     =   32 (&H20)
   DSCAPS_PRIMARY16BIT   =    8
   DSCAPS_PRIMARY8BIT    =    4
   DSCAPS_PRIMARYMONO    =    1
   DSCAPS_PRIMARYSTEREO  =    2
   DSCAPS_SECONDARY16BIT  = 2048 (&H800)
   DSCAPS_SECONDARY8BIT   = 1024 (&H400)
   DSCAPS_SECONDARYMONO   =  256 (&H100)
   DSCAPS_SECONDARYSTEREO =  512 (&H200)
End Enum
```

Constants

DSCAPS_CERTIFIED
This driver has been tested and certified by Microsoft. This flag is always set for WDM drivers, and should not be relied on.

IDH_CONST_DSCAPSFLAGS_dxaudio_vb

DSCAPS_CONTINUOUSRATE
The device supports all sample rates between the **lMinSecondarySampleRate** and **lMaxSecondarySampleRate** member values. Typically, this means that the actual output rate is within 10 hertz (Hz) of the requested frequency.

DSCAPS_EMULDRIVER
The device does not have a DirectSound driver installed, so it is being emulated through the waveform-audio functions. Performance degradation should be expected.

DSCAPS_PRIMARY16BIT
The device supports primary sound buffers with 16-bit samples.

DSCAPS_PRIMARY8BIT
The device supports primary buffers with 8-bit samples.

DSCAPS_PRIMARYMONO
The device supports monophonic primary buffers.

DSCAPS_PRIMARYSTEREO
The device supports stereo primary buffers.

DSCAPS_SECONDARY16BIT
The device supports hardware-mixed secondary sound buffers with 16-bit samples.

DSCAPS_SECONDARY8BIT
The device supports hardware-mixed secondary buffers with 8-bit samples.

DSCAPS_SECONDARYMONO
The device supports hardware-mixed monophonic secondary buffers.

DSCAPS_SECONDARYSTEREO
The device supports hardware-mixed stereo secondary buffers.

CONST_DSCBCAPSFLAGS

[#]Used in the **lFlags** member of the **DSCBCAPS** type to specify device capabilities.

```
Enum CONST_DSCBCAPSFLAGS
   DSCBCAPS_DEFAULT    =          0
   DSCBCAPS_WAVEMAPPED = -2147483648 (&H80000000)
End Enum
```

Constants

DSCBCAPS_WAVEMAPPED
The Win32 wave mapper is used for formats not supported by the device.

[#] IDH_CONST_DSCBCAPSFLAGS_dxaudio_vb

CONST_DSCBLOCKFLAGS

[#]Used by the **DirectSoundCaptureBuffer8.ReadBuffer** and **DirectSoundCaptureBuffer8.WriteBuffer** methods to specify what is read from or written to the sound capture buffer.

```
Enum CONST_DSCBLOCKFLAGS
   DSCBLOCK_DEFAULT     = 0
   DSCBLOCK_ENTIREBUFFER = 1
End Enum
```

Constants

DSCBLOCK_DEFAULT
: Read from or write to the offset specified by the *start* parameter.

DSCBLOCK_ENTIREBUFFER
: Read from or write to the entire buffer. The *size* parameter is ignored. If *start* is not zero, the data transfer begins at *start* and wraps around to the beginning of the buffer.

CONST_DSCBSTARTFLAGS

[#]Used in the *flags* member of the **DirectSoundCaptureBuffer8.Start** method to specify the behavior of the capture buffer when capturing sound data.

```
Enum CONST_DSCBSTARTFLAGS
   DSCBSTART_DEFAULT = 0
   DSCBSTART_LOOPING = 1
End Enum
```

Constants

DSCBSTART_DEFAULT
: Capture stops when the end of the buffer is reached.

DSCBSTART_LOOPING
: Once the end of the buffer is reached, capture restarts at the beginning and continues until explicitly stopped.

IDH_CONST_DSCBLOCKFLAGS_dxaudio_vb
IDH_CONST_DSCBSTARTFLAGS_dxaudio_vb

CONST_DSCBSTATUSFLAGS

#Describe the status of a capture buffer and are returned by the **DirectSoundCaptureBuffer8.GetStatus** method.

```
Enum CONST_DSCBSTATUSFLAGS
   DSCBSTATUS_CAPTURING = 1
   DSCBSTATUS_LOOPING   = 2
End Enum
```

Constants

DSCBSTATUS_CAPTURING
: The capture buffer is capturing sound data.

DSCBSTATUS_LOOPING
: The capture buffer is looping. This value is always returned in combination with DSCBSTATUS_CAPTURING.

CONST_DSCCAPSFLAGS

#Used in the **lFlags** member of the **DSCCAPS** type to describe capture device capabilities.

```
Enum CONST_DSCCAPSFLAGS
   DSCCAPS_DEFAULT    =  0
   DSCCAPS_EMULDRIVER = 32 (&H20)
End Enum
```

Constants

DSCCAPS_DEFAULT
: The device is capable of DirectSound capture.

DSCCAPS_EMULDRIVER
: There is no capable driver for the device, so the standard wave audio functions are being used.

CONST_DSFX_DELAY

#Used in the **DSFXCHORUS** type.

```
Enum CONST_DSFX_DELAY
```

IDH_CONST_DSCBSTATUSFLAGS_dxaudio_vb
IDH_CONST_DSCCAPSFLAGS_dxaudio_vb
IDH_CONST_DSFX_DELAY_dxaudio_vb

```
    DSFX_DELAY_MAX = 20 (&H14)
    DSFX_DELAY_MIN =  0
End Enum
```

Constants

DSFX_DELAY_MAX
Maximum value.

DSFX_DELAY_MIN
Minimum value.

CONST_DSFX_DEPTH

[#]Used in the **DSFXCHORUS** and **DSFXFLANGER** types.

```
Enum CONST_DSFX_DEPTH
    DSFX_DEPTH_MAX  = 100 (&H64)
    DSFX_DEPTH_MIN  =   0
End Enum
```

Constants

DSFX_DEPTH_MAX
Maximum value.

DSFX_DEPTH_MIN
Minimum value.

CONST_DSFXECHO_FEEDBACK

[#]Used in the **DSFXECHO** type.

```
Enum CONST_DSFXECHO_FEEDBACK
    DSFXECHO_FEEDBACK_MAX  = 100 (&H64)
    DSFXECHO_FEEDBACK_MIN  =   0
End Enum
```

Constants

DSFXECHO_FEEDBACK_MAX
Maximum value.

DSFXECHO_FEEDBACK_MIN

IDH_CONST_DSFX_DEPTH_dxaudio_vb
IDH_CONST_DSFXECHO_FEEDBACK_dxaudio_vb

Minimum value.

CONST_DSFX_FEEDBACK

[#]Used in the **DSFXCHORUS** and **DSFXFLANGER** types.

```
Enum CONST_DSFX_FEEDBACK
   DSFX_FEEDBACK_MAX  =  99 (&H63)
   DSFX_FEEDBACK_MIN  = -99 (&HFFFFFF9D)
End Enum
```

Constants

DSFX_FEEDBACK_MAX
Maximum value.

DSFX_FEEDBACK_MIN
Minimum value.

CONST_DSFX_FLANGER_DELAY

[#]Used in the **DSFXFLANGER** type.

```
Enum CONST_DSFX_FLANGER_DELAY
   DSFX_FLANGER_DELAY_MAX =  4
   DSFX_FLANGER_DELAY_MIN =  0
End Enum
```

Constants

DSFX_FLANGER_DELAY_MAX
Maximum value.

DSFX_FLANGER_DELAY_MIN
Minimum value.

CONST_DSFX_FREQUENCY

[#]Used in the **DSFXCHORUS** and **DSFXFLANGER** types.

```
Enum CONST_DSFX_FREQUENCY
   DSFX_FREQUENCY_MAX       = 10
   DSFX_FREQUENCY_MIN       = 0
```

IDH_CONST_DSFX_FEEDBACK_dxaudio_vb
IDH_CONST_DSFX_FLANGER_DELAY_dxaudio_vb
IDH_CONST_DSFX_FREQUENCY_dxaudio_vb

End Enum

Constants

DSFX_FREQUENCY_MAX
Maximum value.

DSFX_FREQUENCY_MIN
Minimum value.

CONST_DSFX_I3DL2_ENVIRONMENT_PRESETS

[#]Used in methods of **DirectSoundFXI3DL2Reverb8** to identify predefined sets of effect parameters.

```
Enum CONST_DSFX_I3DL2_ENVIRONMENT_PRESETS
   DSFX_I3DL2_ENVIRONMENT_PRESET_ALLEY        = 15
   DSFX_I3DL2_ENVIRONMENT_PRESET_ARENA        = 10
   DSFX_I3DL2_ENVIRONMENT_PRESET_AUDITORIUM   =  7
   DSFX_I3DL2_ENVIRONMENT_PRESET_BATHROOM     =  4
   DSFX_I3DL2_ENVIRONMENT_PRESET_CARPETEDHALLWAY=12
   DSFX_I3DL2_ENVIRONMENT_PRESET_CAVE         =  9
   DSFX_I3DL2_ENVIRONMENT_PRESET_CITY         = 17 (&H11)
   DSFX_I3DL2_ENVIRONMENT_PRESET_CONCERTHALL  =  8
   DSFX_I3DL2_ENVIRONMENT_PRESET_DEFAULT      =  0
   DSFX_I3DL2_ENVIRONMENT_PRESET_FOREST       = 16 (&H10)
   DSFX_I3DL2_ENVIRONMENT_PRESET_GENERIC      =  1
   DSFX_I3DL2_ENVIRONMENT_PRESET_HALLWAY      = 13
   DSFX_I3DL2_ENVIRONMENT_PRESET_HANGAR       = 11
   DSFX_I3DL2_ENVIRONMENT_PRESET_LARGEHALL    = 28 (&H1C)
   DSFX_I3DL2_ENVIRONMENT_PRESET_LARGEROOM    = 26 (&H1A)
   DSFX_I3DL2_ENVIRONMENT_PRESET_LIVINGROOM   =  5
   DSFX_I3DL2_ENVIRONMENT_PRESET_MEDIUMHALL   = 27 (&H1B)
   DSFX_I3DL2_ENVIRONMENT_PRESET_MEDIUMROOM   = 25 (&H19)
   DSFX_I3DL2_ENVIRONMENT_PRESET_MOUNTAINS    = 18 (&H12)
   DSFX_I3DL2_ENVIRONMENT_PRESET_PADDEDCELL   =  2
   DSFX_I3DL2_ENVIRONMENT_PRESET_PARKINGLOT   = 21 (&H15)
   DSFX_I3DL2_ENVIRONMENT_PRESET_PLAIN        = 20 (&H14)
   DSFX_I3DL2_ENVIRONMENT_PRESET_PLATE        = 29 (&H1D)
   DSFX_I3DL2_ENVIRONMENT_PRESET_QUARRY       = 19 (&H13)
   DSFX_I3DL2_ENVIRONMENT_PRESET_ROOM         =  3
   DSFX_I3DL2_ENVIRONMENT_PRESET_SEWERPIPE    = 22 (&H16)
   DSFX_I3DL2_ENVIRONMENT_PRESET_SMALLROOM    = 24 (&H18)
```

[#] IDH_CONST_DSFX_I3DL2_ENVIRONMENT_PRESETS_dxaudio_vb

```
    DSFX_I3DL2_ENVIRONMENT_PRESET_STONECORRIDOR = 14
    DSFX_I3DL2_ENVIRONMENT_PRESET_STONEROOM     =  6
    DSFX_I3DL2_ENVIRONMENT_PRESET_UNDERWATER    = 23 (&H17)
End Enum
```

Constants

See Dsound.h for a list of values associated with each preset constant.

CONST_DSFX_I3DL2REVERB_DENSITY

[#]Used in the **fDensity** member of the **DSFXI3DL2REVERB** type.

```
Enum CONST_DSFX_I3DL2REVERB_DENSITY
    DSFX_I3DL2REVERB_DENSITY_DEFAULT = 100 (&H64)
    DSFX_I3DL2REVERB_DENSITY_MAX     = 100 (&H64)
    DSFX_I3DL2REVERB_DENSITY_MIN     =   0
End Enum
```

Constants

DSFX_I3DL2REVERB_DENSITY_DEFAULT
    Default value.

DSFX_I3DL2REVERB_DENSITY_MAX
    Maximum value.

DSFX_I3DL2REVERB_DENSITY_MIN
    Minimum value.

CONST_DSFX_I3DL2REVERB_DIFFUSION

[#]Used in the **fDiffusion** member of the **DSFXI3DL2REVERB** type.

```
Enum CONST_DSFX_I3DL2REVERB_DIFFUSION
    DSFX_I3DL2REVERB_DIFFUSION_DEFAULT = 100 (&H64)
    DSFX_I3DL2REVERB_DIFFUSION_MAX     = 100 (&H64)
    DSFX_I3DL2REVERB_DIFFUSION_MIN     =   0
End Enum
```

IDH_CONST_DSFX_I3DL2REVERB_DENSITY_dxaudio_vb
IDH_CONST_DSFX_I3DL2REVERB_DIFFUSION_dxaudio_vb

Constants

DSFX_I3DL2REVERB_DIFFUSION_DEFAULT
Default value.

DSFX_I3DL2REVERB_DIFFUSION_MAX
Maximum value.

DSFX_I3DL2REVERB_DIFFUSION_MIN
Minimum value.

CONST_DSFX_I3DL2REVERB_HFREFERENCE

[#]Used in the **fHFReference** member of the **DSFXI3DL2REVERB** type.

```
Enum CONST_DSFX_I3DL2REVERB_HFREFERENCE
   DSFX_I3DL2REVERB_HFREFERENCE_DEFAULT =  5000 (&H1388)
   DSFX_I3DL2REVERB_HFREFERENCE_MAX     = 20000 (&H4E20)
   DSFX_I3DL2REVERB_HFREFERENCE_MIN     =    20 (&H14)
End Enum
```

Constants

DSFX_I3DL2REVERB_HFREFERENCE_DEFAULT
Default value.

DSFX_I3DL2REVERB_HFREFERENCE_MAX
Maximum value.

DSFX_I3DL2REVERB_HFREFERENCE_MIN
Minimum value.

CONST_DSFX_I3DL2REVERB_QUALITY

[#]Used in the *lQuality* parameter of the **DirectSoundFXI3DL2Reverb8.SetQuality** method.

```
Enum CONST_DSFX_I3DL2REVERB_REFLECTIONS
   DSFX_I3DL2REVERB_QUALITY_DEFAULT = 2
   DSFX_I3DL2REVERB_QUALITY_MAX     = 3
   DSFX_I3DL2REVERB_QUALITY_MIN     = 0
End Enum
```

IDH_CONST_DSFX_I3DL2REVERB_HFREFERENCE_dxaudio_vb
IDH_CONST_DSFX_I3DL2REVERB_QUALITY_dxaudio_vb

Constants

DSFX_I3DL2REVERB_QUALITY_DEFAULT
Default value.

DSFX_I3DL2REVERB_QUALITY_MAX
Maximum value.

DSFX_I3DL2REVERB_QUALITY_MIN
Minimum value.

CONST_DSFX_I3DL2REVERB_REFLECTIONS

[#]Used in the **lReflectionsDelay** member of the **DSFXI3DL2REVERB** type.

```
Enum CONST_DSFX_I3DL2REVERB_REFLECTIONS
   DSFX_I3DL2REVERB_REFLECTIONS_DEFAULT = -10000
(&HFFFFD8F0)
   DSFX_I3DL2REVERB_REFLECTIONS_MAX     =   1000 (&H3E8)
   DSFX_I3DL2REVERB_REFLECTIONS_MIN     = -10000 (&HFFFFD8F0)
End Enum
```

Constants

DSFX_I3DL2REVERB_REFLECTIONS_DEFAULT
Default value.

DSFX_I3DL2REVERB_REFLECTIONS_MAX
Maximum value.

DSFX_I3DL2REVERB_REFLECTIONS_MIN
Minimum value.

CONST_DSFX_I3DL2REVERB_REVERB

[#]Used in the **lReverbDelay** member of the **DSFXI3DL2REVERB** type.

```
Enum CONST_DSFX_I3DL2REVERB_REVERB
   DSFX_I3DL2REVERB_REVERB_DEFAULT = -10000 (&HFFFFD8F0)
   DSFX_I3DL2REVERB_REVERB_MAX =       2000 (&H7D0)
   DSFX_I3DL2REVERB_REVERB_MIN     = -10000 (&HFFFFD8F0)
```

IDH_CONST_DSFX_I3DL2REVERB_REFLECTIONS_dxaudio_vb
IDH_CONST_DSFX_I3DL2REVERB_REVERB_dxaudio_vb

```
End Enum
```

Constants

DSFX_I3DL2REVERB_REVERB_DEFAULT
Default value.

DSFX_I3DL2REVERB_REVERB_MAX
Maximum value.

DSFX_I3DL2REVERB_REVERB_MIN
Minimum value.

CONST_DSFX_I3DL2REVERB_ROOM

#Used in the **lRoom** member of the **DSFXI3DL2REVERB** type.

```
Enum CONST_DSFX_I3DL2REVERB_ROOM
   DSFX_I3DL2REVERB_ROOM_DEFAULT = -10000 (&HFFFFD8F0)
   DSFX_I3DL2REVERB_ROOM_MAX     =      0
   DSFX_I3DL2REVERB_ROOM_MIN     = -10000 (&HFFFFD8F0)
End Enum
```

Constants

DSFX_I3DL2REVERB_ROOM_DEFAULT
Default value.

DSFX_I3DL2REVERB_ROOM_MAX
Maximum value.

DSFX_I3DL2REVERB_ROOM_MIN
Minimum value.

CONST_DSFX_I3DL2REVERB_ROOMHF

#Used in the **lRoomHF** member of the **DSFXI3DL2REVERB** type.

```
Enum CONST_DSFX_I3DL2REVERB_ROOMHF
   DSFX_I3DL2REVERB_ROOMHF_DEFAULT =      0
   DSFX_I3DL2REVERB_ROOMHF_MAX     =      0
   DSFX_I3DL2REVERB_ROOMHF_MIN     = -10000 (&HFFFFD8F0)
```

IDH_CONST_DSFX_I3DL2REVERB_ROOM_dxaudio_vb
IDH_CONST_DSFX_I3DL2REVERB_ROOMHF_dxaudio_vb

```
End Enum
```

Constants

DSFX_I3DL2REVERB_ROOMHF_DEFAULT
Default value.

DSFX_I3DL2REVERB_ROOMHF_MAX
Maximum value.

DSFX_I3DL2REVERB_ROOMHF_MIN
Minimum value.

CONST_DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR

[#]Used in the **lRoomRolloffFactor** member of the **DSFXI3DL2REVERB** type.

```
Enum CONST_DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR
   DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_DEFAULT = 0
   DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_MAX     = 1
   DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_MIN     = 0
End Enum
```

Constants

DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_DEFAULT
Default value.

DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_MAX
Maximum value.

DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_MIN
Minimum value.

CONST_DSFX_LEFTRIGHTDELAY

[#]Used in the **DSFXECHO** type.

```
Enum CONST_DSFX_LEFTRIGHTDELAY
   DSFX_LEFTRIGHTDELAY_MAX = 2000 (&H7D0)
   DSFX_LEFTRIGHTDELAY_MIN =    1
End Enum
```

IDH_CONST_DSFX_I3DL2REVERB_ROOMROLLOFFFACTOR_dxaudio_vb
IDH_CONST_DSFX_LEFTRIGHTDELAY_dxaudio_vb

Constants

DSFX_LEFTRIGHTDELAY_MAX
Maximum value.

DSFX_LEFTRIGHTDELAY_MIN
Minimum value.

CONST_DSFX_PANDELAY

#Used in the **DSFXECHO** type.

```
Enum CONST_DSFX_PANDELAY
   DSFX_PANDELAY_MAX = 1
   DSFX_PANDELAY_MIN = 0
End Enum
```

Constants

DSFX_PANDELAY_MAX
Maximum value; equivalent to True.

DSFX_PANDELAY_MIN
Minimum value; equivalent to False.

CONST_DSFX_PHASE

#Used in the **DSFXCHORUS** and **DSFXFLANGER** types.

```
Enum CONST_DSFX_PHASE
   DSFX_PHASE_180     = 4
   DSFX_PHASE_90      = 3
   DSFX_PHASE_MAX     = 4
   DSFX_PHASE_MIN     = 0
   DSFX_PHASE_NEG_180 = 0
   DSFX_PHASE_NEG_90  = 1
   DSFX_PHASE_ZERO    = 2
End Enum
```

Constants

DSFX_PHASE_180
Descending midpoint of the waveform.

DSFX_PHASE_90

IDH_CONST_DSFX_PANDELAY_dxaudio_vb
IDH_CONST_DSFX_PHASE_dxaudio_vb

Top of the waveform.

DSFX_PHASE_MAX
Maximum value.

DSFX_PHASE_MIN
Minimum value.

DSFX_PHASE_NEG_180
Ascending midpoint of the waveform.

DSFX_PHASE_NEG_90
Bottom of the waveform.

DSFX_PHASE_ZERO
Ascending midpoint of the waveform.

CONST_DSFX_WAVE

[#]Used in the **DSFXCHORUS** and **DSFXFLANGER** types.

```
Enum CONST_DSFX_WAVE
   DSFX_WAVE_SIN      = 1
   DSFX_WAVE_TRIANGLE  = 0
End Enum
```

Constants

DSFX_WAVE_SIN
Sine wave.

DSFX_WAVE_TRIANGLE
Triangular wave.

CONST_DSFX_WAVESREVERB_INGAIN

[#]Used in the **fInGain** member of the **DSFXWAVESREVERB** type.

```
Enum CONST_DSFX_WAVESREVERB_INGAIN
   DSFX_WAVESREVERB_INGAIN_MAX  =   0
   DSFX_WAVESREVERB_INGAIN_MIN  = -96 (&HFFFFFFA0)
End Enum
```

IDH_CONST_DSFX_WAVE_dxaudio_vb
IDH_CONST_DSFX_WAVESREVERB_INGAIN_dxaudio_vb

Constants

DSFX_WAVESREVERB_INGAIN_MAX
Maximum value.

DSFX_WAVESREVERB_INGAIN_MIN
Minimum value.

CONST_DSFX_WAVESREVERB_REVERBMIX

[#]Used in the **fReverbMix** member of the **DSFXWAVESREVERB** type.

```
Enum CONST_DSFX_WAVESREVERB_REVERBMIX
   DSFX_WAVESREVERB_REVERBMIX_MAX  = 0
   DSFX_WAVESREVERB_REVERBMIX_MIN  = -96 (&HFFFFFFA0)
End Enum
```

Constants

DSFX_WAVESREVERB_REVERBMIX_MAX
Maximum value.

DSFX_WAVESREVERB_REVERBMIX_MIN
Minimum value

CONST_DSFX_WETDRY

[#]Used in the **DSFXCHORUS**, **DSFXECHO**, and **DSFXFLANGER** types.

```
Enum CONST_DSFX_WETDRY
   DSFX_WETDRYMIX_MAX = 100 (&H64)
   DSFX_WETDRYMIX_MIN =   0
End Enum
```

Constants

DSFX_WETDRYMIX_MAX
Maximum value.

DSFX_WETDRYMIX_MIN
Minimum value.

IDH_CONST_DSFX_WAVESREVERB_REVERBMIX_dxaudio_vb
IDH_CONST_DSFX_WETDRY_dxaudio_vb

CONST_DSFXCOMPRESSOR_ATTACK

#Used in the **fAttack** member of the **DSFXCOMPRESSOR** type.

```
Enum CONST_DSFXCOMPRESSOR_ATTACK
   DSFXCOMPRESSOR_ATTACK_MAX = 500 (&H1F4)
   DSFXCOMPRESSOR_ATTACK_MIN =   1
End Enum
```

Constants

DSFXCOMPRESSOR_ATTACK_MAX
Maximum value.

DSFXCOMPRESSOR_ATTACK_MIN
Minimum value.

CONST_DSFXCOMPRESSOR_GAIN

#Used in the **fGain** member of the **DSFXCOMPRESSOR** type.

```
Enum CONST_DSFXCOMPRESSOR_GAIN
   DSFXCOMPRESSOR_GAIN_MAX =  60 (&H3C)
   DSFXCOMPRESSOR_GAIN_MIN = -60 (&HFFFFFFC4)
End Enum
```

Constants

DSFXCOMPRESSOR_GAIN_MAX
Maximum value.

DSFXCOMPRESSOR_GAIN_MIN
Minimum value.

CONST_DSFXCOMPRESSOR_PREDELAY

#Used in the **fPreDelay** member of the **DSFXCOMPRESSOR** type.

IDH_CONST_DSFXCOMPRESSOR_ATTACK_dxaudio_vb
IDH_CONST_DSFXCOMPRESSOR_GAIN_dxaudio_vb
IDH_CONST_DSFXCOMPRESSOR_PREDELAY_dxaudio_vb

```
Enum CONST_DSFXCOMPRESSOR_PREDELAY
   DSFXCOMPRESSOR_PREDELAY_MAX = 4
   DSFXCOMPRESSOR_PREDELAY_MIN = 0
End Enum
```

Constants

DSFXCOMPRESSOR_PREDELAY_MAX
Maximum value.

DSFXCOMPRESSOR_PREDELAY_MIN
Minimum value.

CONST_DSFXCOMPRESSOR_RATIO

[#]Used in the **fRatio** member of the **DSFXCOMPRESSOR** type.

```
Enum CONST_DSFXCOMPRESSOR_RATIO
   DSFXCOMPRESSOR_RATIO_MAX =  100 (&H64)
   DSFXCOMPRESSOR_RATIO_MIN =    1
End Enum
```

Constants

DSFXCOMPRESSOR_RATIO_MAX
Maximum value.

DSFXCOMPRESSOR_RATIO_MIN
Minimum value.

CONST_DSFXCOMPRESSOR_RELEASE

[#]Used in the **fRelease** member of the **DSFXCOMPRESSOR** type.

```
Enum CONST_DSFXCOMPRESSOR_RELEASE
   DSFXCOMPRESSOR_RELEASE_MAX = 3000 (&HBB8)
   DSFXCOMPRESSOR_RELEASE_MIN =   50 (&H32)
End Enum
```

[#] IDH_CONST_DSFXCOMPRESSOR_RATIO_dxaudio_vb
[#] IDH_CONST_DSFXCOMPRESSOR_RELEASE_dxaudio_vb

Constants

DSFXCOMPRESSOR_RELEASE_MAX
Maximum value.

DSFXCOMPRESSOR_RELEASE_MIN
Minimum value.

CONST_DSFXCOMPRESSOR_THRESHOLD

[#]Used in the **fThreshold** member of the **DSFXCOMPRESSOR** type.

```
Enum CONST_DSFXCOMPRESSOR_THRESHOLD
   DSFXCOMPRESSOR_THRESHOLD_MAX =   0
   DSFXCOMPRESSOR_THRESHOLD_MIN = -60 (&HFFFFFFC4)
End Enum
```

Constants

DSFXCOMPRESSOR_THRESHOLD_MAX
Maximum value.

DSFXCOMPRESSOR_THRESHOLD_MIN
Minimum value.

CONST_DSFXDISTORTION_EDGE

[#]Used in the **DSFXDISTORTION** type.

```
Enum CONST_DSFXDISTORTION_EDGE
   DSFXDISTORTION_EDGE_MAX = 100 (&H64)
   DSFXDISTORTION_EDGE_MIN =   0
End Enum
```

Constants

DSFXDISTORTION_EDGE_MAX
Maximum value.

DSFXDISTORTION_EDGE_MIN
Minimum value.

IDH_CONST_DSFXCOMPRESSOR_THRESHOLD_dxaudio_vb
IDH_CONST_DSFXDISTORTION_EDGE_dxaudio_vb

CONST_DSFXDISTORTION_GAIN

[#]Used in the **DSFXDISTORTION** type.

```
Enum CONST_DSFXDISTORTION_GAIN
   DSFXDISTORTION_GAIN_MAX =   0
   DSFXDISTORTION_GAIN_MIN = -60 (&HFFFFFFC4)
End Enum
```

Constants

DSFXDISTORTION_GAIN_MAX
: Maximum value.

DSFXDISTORTION_GAIN_MIN
: Minimum value.

CONST_DSFXDISTORTION_POSTEQBANDWIDTH

[#]Used in the **DSFXDISTORTION** type.

```
Enum CONST_DSFXDISTORTION_POSTEQBANDWIDTH
   DSFXDISTORTION_POSTEQBANDWIDTH_MAX = 8000 (&H1F40)
   DSFXDISTORTION_POSTEQBANDWIDTH_MIN =  100 (&H64)
End Enum
```

Constants

DSFXDISTORTION_POSTEQBANDWIDTH_MAX
: Maximum value.

DSFXDISTORTION_POSTEQBANDWIDTH_MIN
: Maximum value.

CONST_DSFXDISTORTION_POSTEQCENTERFREQUENCY

[#]Used in the **DSFXDISTORTION** type.

IDH_CONST_DSFXDISTORTION_GAIN_dxaudio_vb
IDH_CONST_DSFXDISTORTION_POSTEQBANDWIDTH_dxaudio_vb
IDH_CONST_DSFXDISTORTION_POSTEQCENTERFREQUENCY_dxaudio_vb

```
Enum CONST_DSFXDISTORTION_POSTEQCENTERFREQUENCY
   DSFXDISTORTION_POSTEQCENTERFREQUENCY_MAX = 8000
(&H1F40)
   DSFXDISTORTION_POSTEQCENTERFREQUENCY_MIN = 100 (&H64)
End Enum
```

Constants

DSFXDISTORTION_POSTEQCENTERFREQUENCY_MAX
Maximum value.

DSFXDISTORTION_POSTEQCENTERFREQUENCY_MIN
Minimum value.

CONST_DSFXDISTORTION_PRELOWPASSCUTOFF

#Used in the **DSFXDISTORTION** type.

```
Enum CONST_DSFXDISTORTION_PRELOWPASSCUTOFF
   DSFXDISTORTION_PRELOWPASSCUTOFF_MAX = 8000 (&H1F40)
   DSFXDISTORTION_PRELOWPASSCUTOFF_MIN =  100 (&H64)
End Enum
```

Constants

DSFXDISTORTION_PRELOWPASSCUTOFF_MAX
Maximum value.

DSFXDISTORTION_PRELOWPASSCUTOFF_MIN
Minimum value.

CONST_DSFXGARGLE_RATEHZ

#Used in the **DSFXGARGLE** type.

```
Enum CONST_DSFXGARGLE_RATEHZ
   DSFXGARGLE_RATEHZ_MAX = 1000 (&H3E8)
   DSFXGARGLE_RATEHZ_MIN =    1
End Enum
```

IDH_CONST_DSFXDISTORTION_PRELOWPASSCUTOFF_dxaudio_vb
IDH_CONST_DSFXGARGLE_RATEHZ_dxaudio_vb

Constants

DSFXGARGLE_RATEHZ_MAX
Maximum value.

DSFXGARGLE_RATEHZ_MIN
Minimum value.

CONST_DSFXGARGLE_WAVE

#Used in the **DSFXGARGLE** type.

```
Enum CONST_DSFXGARGLE_WAVE
   DSFXGARGLE_WAVE_SQUARE   = 1
   DSFXGARGLE_WAVE_TRIANGLE = 0
End Enum
```

Constants

DSFXGARGLE_WAVE_TRIANGLE
Triangular wave.

DSFXGARGLE_WAVE_SQUARE
Square wave.

CONST_DSFXPARAMEQ_BANDWIDTH

#Used in the **fBandwidth** member of the **DSFXPARAMEQ** type.

```
Enum CONST_DSFXPARAMEQ_BANDWIDTH
   DSFXPARAMEQ_BANDWIDTH_MAX = 36 (&H24)
   DSFXPARAMEQ_BANDWIDTH_MIN =  1
End Enum
```

Constants

DSFXPARAMEQ_BANDWIDTH_MAX
Maximum value.

DSFXPARAMEQ_BANDWIDTH_MIN
Minimum value.

IDH_CONST_DSFXGARGLE_WAVE_dxaudio_vb
IDH_CONST_DSFXPARAMEQ_BANDWIDTH_dxaudio_vb

CONST_DSFXPARAMEQ_CENTER

#Used in the **fCenter** member of the **DSFXPARAMEQ** type.

```
Enum CONST_DSFXPARAMEQ_CENTER
   DSFXPARAMEQ_CENTER_MAX = 16000 (&H3E80)
   DSFXPARAMEQ_CENTER_MIN =    80 (&H50)
End Enum
```

Constants

DSFXPARAMEQ_CENTER_MAX
Maximum value.

DSFXPARAMEQ_CENTER_MIN
Minimum value.

CONST_DSFXPARAMEQ_GAIN

#Used in the **fGain** member of the **DSFXPARAMEQ** type.

```
Enum CONST_DSFXPARAMEQ_GAIN
   DSFXPARAMEQ_GAIN_MAX =  15
   DSFXPARAMEQ_GAIN_MIN = -15 (&HFFFFFFF1)
End Enum
```

Constants

DSFXPARAMEQ_GAIN_MAX
Maximum value.

DSFXPARAMEQ_GAIN_MIN
Minimum value.

CONST_DSOUND

#Miscellaneous constants used throughout DirectSound.

```
Enum
   DS3D_DEFAULTCONEANGLE        = 360 (&H168)
   DS3D_DEFAULTCONEOUTSIDEVOLUME =         0
   DS3D_DEFAULTDISTANCEFACTOR    =         1
   DS3D_DEFAULTDOPPLERFACTOR     =         1
   DS3D_DEFAULTMAXDISTANCE       = 1000000000 (&H3B9ACA00)
```

IDH_CONST_DSFXPARAMEQ_CENTER_dxaudio_vb
IDH_CONST_DSFXPARAMEQ_GAIN_dxaudio_vb
IDH_CONST_DSOUND_dxaudio_vb

```
   DS3D_DEFAULTMINDISTANCE      =         1
   DS3D_DEFAULTROLLOFFFACTOR    =         1
   DS3D_MAXCONEANGLE            =       360 (&H168)
   DS3D_MAXDISTANCEFACTOR       =        10
   DS3D_MAXDOPPLERFACTOR        =        10
   DS3D_MAXROLLOFFFACTOR        =        10
   DS3D_MINCONEANGLE            =         0
   DS3D_MINDISTANCEFACTOR       =         0
   DS3D_MINDOPPLERFACTOR        =         0
   DS3D_MINROLLOFFFACTOR        =         0
   DSBFREQUENCY_MAX             =     100000 (&H186A0)
   DSBFREQUENCY_MIN             =       100 (&H64)
   DSBFREQUENCY_ORIGINAL        =         0
   DSBPAN_CENTER                =         0
   DSBPAN_LEFT                  =    -10000 (&HFFFFD8F0)
   DSBPAN_RIGHT                 =     10000 (&H2710)
   DSBPN_OFFSETSTOP             =        -1 (&HFFFFFFFF)
   DSBSIZE_MAX                  =  268435455 (&HFFFFFFF)
   DSBSIZE_MIN                  =         4
   DSBVOLUME_MAX                =         0
   DSBVOLUME_MIN                =    -10000 (&HFFFFD8F0)
   WAVE_FORMAT_PCM              =         1
End Enum
```

Constants

DS3D_DEFAULTCONEANGLE = 360 (&H168)
DS3D_DEFAULTCONEOUTSIDEVOLUME
DS3D_DEFAULTDISTANCEFACTOR
DS3D_DEFAULTDOPPLERFACTOR
DS3D_DEFAULTMAXDISTANCE
DS3D_DEFAULTMINDISTANCE
DS3D_DEFAULTROLLOFFFACTOR
DS3D_MAXCONEANGLE
DS3D_MAXDISTANCEFACTOR
DS3D_MAXDOPPLERFACTOR
DS3D_MAXROLLOFFFACTOR
DS3D_MINCONEANGLE
DS3D_MINDISTANCEFACTOR
DS3D_MINDOPPLERFACTOR
DS3D_MINROLLOFFFACTOR
Allowable and default values for 3-D buffer parameters.

DSBFREQUENCY_MAX
DSBFREQUENCY_MIN
DSBFREQUENCY_ORIGINAL
DSBPAN_CENTER

DSBPAN_LEFT
DSBPAN_RIGHT
DSBPN_OFFSETSTOP
DSBSIZE_MAX
DSBSIZE_MIN
DSBVOLUME_MAX
DSBVOLUME_MIN
Allowable and default values for buffer parameters.

WAVE_FORMAT_PCM
Used in the **nFormatTag** member of the **WAVEFORMATEX** type.

CONST_DSOUNDERR

#Represent DirectMusic error codes. For a list of values, see DirectSound Error Codes.

CONST_DSOUNDFX

#Used in specifying and retrieving parameters of effects buffer creation.

```
Enum CONST_DSOUNDFX
    DSFX_LOCHARDWARE  = 1
    DSFX_LOCSOFTWARE  = 2
    DSFXR_FAILED      = 4
    DSFXR_LOCHARDWARE = 1
    DSFXR_LOCSOFTWARE = 2
    DSFXR_PRESENT     = 0
    DSFXR_UNALLOCATED = 3
    DSFXR_UNKNOWN     = 5
End Enum
```

Constants

DSFX_LOCHARDWARE
DSFX_LOCSOFTWARE
These flags control the creation of play buffer effects. See **DSEFFECTDESC**.

DSFXR_FAILED
DSFXR_LOCHARDWARE
DSFXR_LOCSOFTWARE
DSFXR_PRESENT
DSFXR_UNALLOCATED
DSFXR_UNKNOWN

IDH_CONST_DSOUNDERR_dxaudio_vb
IDH_CONST_DSOUNDFX_dxaudio_vb

These values describe the result of an attempt to create a play buffer effect. See **DirectSoundSecondaryBuffer8.AcquireResources** and **DirectSoundSecondaryBuffer8.SetFX**.

CONST_DSSCLFLAGS

#Used in the *level* parameter of the **DirectSound8.SetCooperativeLevel** method to specify the priority level.

```
Enum CONST_DSSCLFLAGS
   DSSCL_NORMAL      = 1
   DSSCL_PRIORITY    = 2
   DSSCL_WRITEPRIMARY = 4
End Enum
```

Constants

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

DSSCL_PRIORITY
: Sets the application to the priority level. Applications with this cooperative level can call the **DirectSoundPrimaryBuffer8.SetFormat** method.

DSSCL_WRITEPRIMARY
: Sets the application to the highest priority level. The application has write access to the primary sound buffer.

Remarks

The DSSCL_EXCLUSIVE cooperative level is no longer supported.

CONST_DSSPEAKERFLAGS

#Used by the **DirectSound8.GetSpeakerConfig** and **DirectSound8.SetSpeakerConfig** method to specify the speaker configuration.

```
Enum CONST_DSSPEAKERFLAGS
   DSSPEAKER_GEOMETRY_MAX    = 180 (&HB4)
   DSSPEAKER_GEOMETRY_MIN    =  5
   DSSPEAKER_GEOMETRY_NARROW =  10
   DSSPEAKER_GEOMETRY_WIDE   =  20 (&H14)
   DSSPEAKER_HEADPHONE       =  1
```

IDH_CONST_DSSCLFLAGS_dxaudio_vb
IDH_CONST_DSSPEAKERFLAGS_dxaudio_vb

```
    DSSPEAKER_MONO         =  2
    DSSPEAKER_QUAD         =  3
    DSSPEAKER_STEREO       =  4
    DSSPEAKER_SURROUND     =  5
End Enum
```

Constants

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

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

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

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

DSSPEAKER_HEADPHONE
The speakers are headphones.

DSSPEAKER_MONO
The speakers are monaural.

DSSPEAKER_QUAD
The speakers are quadraphonic.

DSSPEAKER_STEREO
The speakers are stereo.

DSSPEAKER_SURROUND
The speakers are surround sound.

CONST_WAVEFORMATFLAGS

[#]Used in the **lFormats** member of the **DSCCAPS** type to indicate what standard formats are supported.

```
Enum CONST_WAVEFORMATFLAGS
    WAVE_FORMAT_1M08 =    1
    WAVE_FORMAT_1M16 =    4
    WAVE_FORMAT_1S08 =    2
    WAVE_FORMAT_1S16 =    8
    WAVE_FORMAT_2M08 =   16 (&H10)
    WAVE_FORMAT_2M16 =   64 (&H40)
    WAVE_FORMAT_2S08 =   32 (&H20)
    WAVE_FORMAT_2S16 =  128 (&H80)
    WAVE_FORMAT_4M08 =  256 (&H100)
```

IDH_CONST_WAVEFORMATFLAGS_dxaudio_vb

```
    WAVE_FORMAT_4M16 = 1024 (&H400)
    WAVE_FORMAT_4S08 =  512 (&H200)
    WAVE_FORMAT_4S16 = 2048 (&H800)
End Enum
```

Constants

WAVE_FORMAT_1M08
    11.025 kHz, mono, 8-bit.

WAVE_FORMAT_1M16
    11.025 kHz, mono, 16-bit.

WAVE_FORMAT_1S08
    11.025 kHz, stereo, 8-bit.

WAVE_FORMAT_1S16
    11.025 kHz, stereo, 16-bit.

WAVE_FORMAT_2M08
    22.05 kHz, mono, 8-bit.

WAVE_FORMAT_2M16
    22.05 kHz, mono, 16-bit.

WAVE_FORMAT_2S08
    22.05 kHz, stereo, 8-bit.

WAVE_FORMAT_2S16
    22.05 kHz, stereo, 16-bit.

WAVE_FORMAT_4M08
    44.1 kHz, mono, 8-bit.

WAVE_FORMAT_4M16
    44.1 kHz, mono, 16-bit.

WAVE_FORMAT_4S08
    44.1 kHz, stereo, 8-bit.

WAVE_FORMAT_4S16
    44.1 kHz, stereo, 16-bit.

DirectSound Error Codes

This section lists values specific to DirectSound that can be returned in **Err.Number** when an error is raised on a method call. For a list of the specific codes that each method can return, see the individual method descriptions. These lists are not necessarily comprehensive.

Error codes are presented in the following sections:

- DirectSound Error Codes By Number

- DirectSound Error Codes By Name

DirectSound Error Codes By Number

The following table lists DirectSound error codes sorted by value. For a description, click on the constant.

Hexadecimal	Constant
&H00000007	DSERR_OUTOFMEMORY
&H000001AE	DSERR_NOINTERFACE
&H80004001	DSERR_UNSUPPORTED
&H80004005	DSERR_GENERIC
&H80070057	DSERR_INVALIDPARAM
&H8878000A	DSERR_ALLOCATED
&H8878001E	DSERR_CONTROLUNAVAIL
&H88780032	DSERR_INVALIDCALL
&H88780046	DSERR_PRIOLEVELNEEDED
&H88780064	DSERR_BADFORMAT
&H88780078	DSERR_NODRIVER
&H88780082	DSERR_ALREADYINITIALIZED
&H88780096	DSERR_BUFFERLOST
&H887800A0	DSERR_OTHERAPPHASPRIO
&H887800AA	DSERR_UNINITIALIZED
&H887810B4	DSERR_BUFFERTOOSMALL
&H887810BE	DSERR_DS8_REQUIRED
&H88781161	DSERR_OBJECTNOTFOUND

DirectSound Error Codes By Name

The following list describes all DirectSound error codes. To find a constant from its value, see DirectSound Error Codes By Number.

DSERR_ALLOCATED
: The request failed because resources, such as a priority level, were in use by another caller.

DSERR_ALREADYINITIALIZED
: The object is already initialized.

DSERR_BADFORMAT
: The specified wave format is not supported.

DSERR_BUFFERLOST
: The buffer memory has been lost and must be restored.

DSERR_BUFFERTOOSMALL
The buffer is not large enough to support effects processing.

DSERR_CONTROLUNAVAIL
The control (volume, pan, and so forth) requested by the caller is not available.

DSERR_DS8_REQUIRED
A DirectSound object of class CLSID_DirectSound8 or later is required for the requested functionality.

DSERR_GENERIC
An undetermined error occurred inside the DirectSound subsystem.

DSERR_INVALIDCALL
This function is not valid for the current state of this object.

DSERR_INVALIDPARAM
An invalid parameter was passed to the returning function.

DSERR_NODRIVER
No sound driver is available for use.

DSERR_NOINTERFACE
The requested COM interface is not available.

DSERR_OBJECTNOTFOUND
The requested object was not found.

DSERR_OTHERAPPHASPRIO
Another application has a higher priority level, preventing this call from succeeding.

DSERR_OUTOFMEMORY
The DirectSound subsystem could not allocate sufficient memory to complete the caller's request.

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

DSERR_UNINITIALIZED
The DirectSound device has not been initialized.

DSERR_UNSUPPORTED
The function called is not supported at this time.