DirectMusic

This section provides information about using Microsoft® DirectMusic® to capture and play sounds in applications written in C and C++.

Information is presented in the following sections:

- <u>What's New in DirectMusic</u>. A guide to new features of DirectMusic in Microsoft DirectX® 9.0.
- Introduction to DirectMusic. An overview of DirectMusic: its capabilities, basic concepts, and architecture, together with an introduction to dynamic soundtracks. This section focuses on principles rather than on the practical side of programming for DirectMusic.
- <u>Getting Started with DirectMusic</u>. Information on setting up and debugging DirectMusic projects, and an overview of the programming steps involved in setting up a DirectMusic performance and playing a sound.
- <u>Using DirectMusic</u>. A guide to using the DirectMusic application programming interface (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. It should be used in conjunction with the reference section.
- <u>Advanced Topics in DirectMusic</u>. Information of interest mostly for developing specialized applications or applications that need highly optimized performance.
- <u>Related Software</u>. Information about software distributed with the DirectX SDK that can be used in conjunction with DirectMusic.
- <u>DirectMusic C++ Samples</u>. 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 DirectMusic.
- <u>DirectMusic C++ Tutorials</u>. Step-by-step guides to implementing basic DirectMusic functionality.
- <u>DirectMusic C/C++ Reference</u>. Detailed information about all the API elements declared in the DirectMusic header files.

^{© 2004} Microsoft Corporation. All rights reserved.

What's New in DirectMusic

The DirectMusic application programming interface (API) has not been significantly revised for DirectX 9.0. However, many performance enhancements have been made. The most significant of these is a new low-latency DirectSound sink, which enables DirectMusic to attain much quicker response when using audiopaths that play through the software synthesizer. This enhancement is of particular interest to sound designers and composers who want to take advantage of the rich DirectMusic feature set but who also require low latency for sound effects. For more information, see <u>Reducing Latency</u>.

Several new features have been added to content created in DirectMusic Producer. For example, streamed waves in wave tracks can now be looped for the easier creation of ambient sounds, and looping can be done in clock time so that looping waves are not affected by tempo changes. For more information, see What's New in This Release in the DirectMusic Producer Help.

A large library of DirectMusic styles has been added to the SDK, along with an application for auditioning them.

In addition, two new tools have been made available in the \bin\DXUtils\AppWizard folder of the SDK installation:

- DMToolWizard.awx is a wizard for creating DirectMusic tools.
- AEDMOWiz.awx is a wizard for creating effect <u>DMOs</u>.

See Also

<u>Related Software</u>

Introduction to DirectMusic

This section provides a quick overview of the capabilities of DirectMusic and what you need to do to get started with the API. Information is presented in the following sections:

- The Power of DirectMusic
- <u>Elements of a DirectMusic Application</u>
- Overview of Audio Data Flow
- Introduction to Dynamic Musical Soundtracks

The Power of DirectMusic

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

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

- Load and play sounds from files or resources in MIDI, WAV, 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 DLS, 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, reverberation, and other effects.
- Use more than 16 MIDI channels. DirectMusic 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.

If you use source files from DirectMusic Producer, 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 waveforms with variations.
- Map <u>performance channels</u> to different buffers within an audiopath, 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 elements 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.

These capabilities are the ones most often used by mainstream applications. DirectMusic 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.

DirectMusic delivers full functionality on Microsoft Windows® 98 and Microsoft Windows 2000. Support for hardware synthesizers is available only on Windows 98 Second Edition, Windows 2000, and later operating systems. The default Microsoft software synthesizer is always available.

Although DirectMusic loads and plays WAV files, applications that need highly optimized performance or low-level control over sound buffers can still use the DirectSound API. For a comparison of the two APIs, see the DirectX documentation.

^{© 2004} Microsoft Corporation. All rights reserved.

Elements of a DirectMusic Application

This section is an introduction to some of the key concepts and code objects of DirectMusic. Although some interfaces are introduced, this section does not get into the details of using the API. For practical information on writing DirectMusic applications, see <u>Using DirectMusic</u> or the specific topics listed under See Also at the end of each topic.

The following topics are discussed:

- <u>Loader</u>
- <u>Segments and Segment States</u>
- <u>Performance</u>
- <u>Messages</u>
- Performance Channels
- Downloadable Sounds
- Instruments and Downloading
- Audiopaths and Buffers
- Audio Scripts

Loader

The loader, represented by the **IDirectMusicLoader8** interface, is an object that gets other objects. It is generally one of the first objects created in a DirectMusic application. The loader is used to load all audio content, including DirectMusic segment files, <u>DLS</u> collections, MIDI files, and WAV files. It can also load data stored in resources or application memory.

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**.

See Also

• Loading Audio Data

Segments and Segment States

Segments are objects encapsulating sequenced sound data. The data might be a MIDI sequence, a waveform, 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 WAV 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.

Segments can combine different kinds of data such as waveforms, 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.

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.

See Also

• <u>Using Segments</u>

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.

See Also

• <u>Creating the Performance</u>

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 waveform, or a controller change. It might even contain text for a display of lyrics.

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. It is also possible to intercept messages by using plug-in components called tools.

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.

See Also

- Notification and Event Handling
- <u>Using DirectMusic Messages</u>

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 waveform. Often a part corresponds to a single musical instrument.

A performance channel controls the route taken by a part through an audiopath. It maps the part to a resource on the synthesizer (an instrument timbre or a waveform) and also enables the output of the synthesizer for the part to be sent through a particular configuration of buffers for the application of effects. Every message that contains information about a part also specifies the part's performance channel, so that it can be routed correctly. Every performance channel has its own settings for pan, volume, and transposition.

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.

Note The array of performance channels in the DirectMusic API is zero-based, but in DirectMusic Producer it is one-based. Performance channel 1 in DirectMusic Producer is performance channel 0 in the API.

See Also

• <u>Using Bands</u>

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 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, which is the synthesis of instrument sounds from digital samples. These samples are obtained from recordings of real instruments and stored on the hardware. The samples are looped and modulated to produce sounds of any length at different pitches and volumes.

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

The Downloadable Sounds (DLS) standard, published by the MIDI Manufacturers Association, is a way of enabling 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, developers 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 modulated 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, or shapes, 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*. Individual instruments are assigned patch numbers and respond to MIDI messages just like traditional MIDI instruments. However, a DLS instrument does not have to belong to the General MIDI set. 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.

By default, DirectMusic uses the Microsoft software synthesizer, which supports DLS Level 2.

Note The DLS Level 1 synthesizer used with the DirectX 7 interfaces contains reverberation 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 later interfaces does not contain built-in reverberation capabilities. Reverberation is instead implemented as a

<u>DirectX Media Object</u>. Waves MaxxVerb is licensed to Microsoft Corporation for this purpose.

See Also

• <u>Using Bands</u>

Instruments and Downloading

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

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 waveform 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 <u>performance channels</u>. Several techniques are available for ensuring that band instruments are downloaded before use.

Note The Roland GM/GS Sound Set cannot be modified, due to legal restrictions.

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

A *band* is a collection of <u>performance channel</u> settings that determine how parts in a sound file are played. For each channel, a band includes an instrument assignment as well as settings for volume, pan, and transposition. Every part in a performance is mapped to a performance channel, and each part plays with the instrument settings applied to its channel by the band.

Although a band can be thought of as a set of instruments, it is not the same as a DLS collection. A DLS collection is a set of instruments that can be downloaded to the synthesizer and thus made available to any application. The instruments themselves contain no information about assignment to performance channels, volume, pan, or transposition. This data must be supplied by bands for the

channels used by a performance.

Note When bands are downloaded to the synthesizer, it is really the DLS instruments referenced by the band that are being downloaded.

Bands can be saved as separate files or included in <u>styles</u> or segments.

See Also

• <u>Using Bands</u>

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.

Note The buffers referred to here are used for streaming and processing <u>PCM</u> data after it has left the synthesizer, and these buffers 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 WAV 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 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 <u>DMO</u> to change the parameters of the effect.

See Also

• Using Audiopaths

Audio Scripts

An audio script is a file containing variables and routines that can be accessed from within an application. Scripts can be written using any Microsoft ActiveX® Scripting language, but DirectMusic Producer also enables scripting in a special language called AudioVBScript. The advantage of using AudioVBScript is that it requires a very small run-time library, which is one of the DirectMusic system components. AudioVBScript is a subset of Microsoft Visual Basic® Scripting Edition.

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

Scripts implement some of the key DirectMusic 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

DirectMusic scripts make it easier for application developers and sound designers to coordinate their efforts. Scripts give sound designers 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.

The following scenario is one in which scripting might be helpful. It supposes that the sound effects for a game are stored as individual WAV 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 times in the game. But now the sound designers want to make some changes. They decide, for example, that the boss monster should have a different grunt than normal monsters. The sound designers 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.

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 play a grunt. Assume that that *szGrunter* has been set to a string constant such as "Player", "Boss", or "NormalMonster", 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, who is probably a member of the sound design team, decides what the routine does. For example, the routine 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 in the existing application.

The scripting API is documented in the DirectMusic Producer Help file.

See Also

• <u>Using Audio Scripts</u>

Overview of Audio Data Flow

Typically, a DirectMusic application obtains sounds from one or more of the following sources:

- MIDI files
- WAV 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 at run time by the DirectMusic 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 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 WAV 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. A tool can modify a message and pass it on, delete it, or send a new message. Tools are arranged in linear sets called toolgraphs. A message might pass through any or all of the following toolgraphs, in the order given:

- Segment toolgraph. Processes messages from a single segment.
- Audiopath toolgraph. Processes messages on a single audiopath.
- Performance toolgraph. Processes all messages in the performance.

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

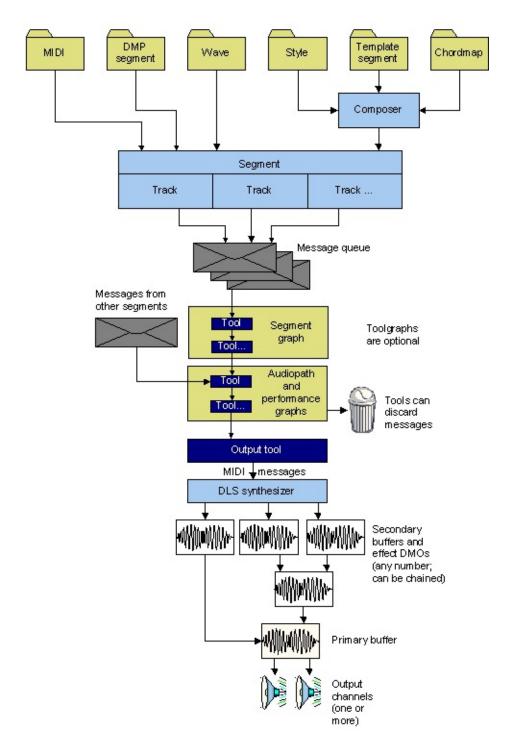
messages are directed to the appropriate channel group on the synthesizer. The synthesizer creates waveforms and streams them to a device called a sink, which manages the distribution of data through buses to DirectSound buffers.

There are three kinds of DirectSound buffers:

- *Sink-in buffers* are DirectSound secondary buffers into which the sink streams data. These buffers enable the application to control pan, volume, 3-D location, 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 waveforms. These buffers can be used to apply global effects. An effect achieved by directing data to a mix-in buffer is called a *send*. Mix-in buffers can be created only by using audiopath configurations authored in DirectMusic Producer.
- The *primary buffer* performs the final mixing on all data and passes it to the rendering device.

Note Applications are not responsible for managing 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, write to it, start it, or stop it by using the **IDirectSoundBuffer8** interface.

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 WAV file, a MIDI file, a segment file authored in DirectMusic Producer, or component files combined by the composer object. In all cases, data can come from a resource rather than a file.



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

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

Introduction to Dynamic Musical Soundtracks

If you want to take full advantage of DirectMusic, you won't play just MIDI and WAV files. You'll take elements authored in DirectMusic Producer 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 the elements of a dynamic soundtrack is meant to give the application developer an understanding of the material being used in the performance. For a more detailed view, see the Help for DirectMusic Producer. For information on how to incorporate these elements in an application, see <u>Using Compositional Elements</u>.

The following topics are covered:

- <u>Styles</u>
- <u>Chordmaps</u>
- <u>Style-based Segments</u>
- <u>Templates</u>
- How Music Varies During Playback
- Music Values and MIDI Notes

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 defined as fixed pitches; they are described according to the role they play in the chord structure. The notes are mapped to MIDI values when they are played, after the current key, chord, and play mode have been taken into account.

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

Each pattern can have up to 32 variations. At run time, variations are chosen by the style playback mechanism. However, the author can specify that a variation must never be chosen when a certain chord is being played.

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

Any pattern (other than a motif) can be designated as an *embellishment*. There are four standard embellishment types: intro, fill, break, and end. In addition, DirectMusic Producer enables authors to establish up to 100 custom embellishment types. A pattern can be assigned to one or more of these types. When a style-based segment is played and a certain type of embellishment is specified at some point in the segment, only patterns of that type are candidates for playback.

See Also

- <u>DirectMusic Style Library</u>
- <u>Style-based Segments</u>

Chordmaps

Much modern music, especially music in the popular, rock, folk, and jazz idioms, is based on the concept of chord progression. 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 <u>pattern</u> 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 modulates the pitch of the notes accordingly.

A *chordmap* is a collection of chords that provides multiple potential chord progressions to a musical piece. Chord progressions are generated from a chordmap and inserted into the chord track of a segment, either at design time or at run time.

By using chordmaps, the author of the music can create multiple segments from a common set of chords. Chordmaps can also be used by the application at run time to create new segments or to build new chord progressions in existing segments.

Certain important chords in a chordmap 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.

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 <u>style</u>.

See Also

• Using Chordmaps

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 <u>patterns</u>.
- *Chord progression*. This might be derived automatically by the authoring tool from a <u>chordmap</u> (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 <u>groove level</u> of 25 for the next four measures, then play a break, and so on.
- *Band*. The author can assign instruments and <u>performance channels</u> to all the parts in the various patterns.

See Also

- <u>Chordmaps</u>
- <u>Styles</u>

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 that contains the track or for another segment that does not have a signpost track. The command track supplies <u>groove levels</u> and embellishments. You can use templates at run time to compose new material.

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 <u>chordmap</u>. 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 <u>Using Templates</u>.

How Music Varies During Playback

As DirectMusic plays a <u>style</u>-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 <u>patterns</u>, 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 <u>groove level</u>. (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 start and stop times of individual notes. The degree of randomness is set by the author of the content.
- *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.

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 change 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 <u>pattern</u> in a DirectMusic <u>style</u> or pattern track are not notes with a fixed MIDI value. Rather, they are *music values* that become MIDI 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 be made up of 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.

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.

See Also

• DMUS_NOTE_PMSG

- <u>DMUS_PLAYMODE_FLAGS</u>
- <u>DMUS_CHORD_PARAM</u>
- <u>DMUS_CHORD_KEY</u>
- <u>DMUS_SUBCHORD</u>
- <u>IDirectMusicPerformance8::MIDIToMusic</u>
- IDirectMusicPerformance8::MusicToMIDI
- <u>DirectMusic Tools</u>

Getting Started with DirectMusic

This section gives information on setting up and debugging DirectMusic projects, as well as a brief overview of the programming steps required to set up a performance and play sounds.

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

- Building DirectMusic Projects
- <u>Debugging DirectMusic Projects</u>
- First Steps in DirectMusic Programming

Building DirectMusic Projects

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.

The following additional headers are not needed by most applications:

- Dmusicf.h: file structures and definitions. This header is needed only by applications such as music-authoring tools that work directly with files and don't rely solely on the loaders built into DirectMusic.
- Dmusics.h: declarations for the **IDirectMusicSynth** and **IDirectMusicSynthSink** interfaces, which are used for creating synthesizers and synthesizer sinks.
- Dmusbuff.h: declaration of the **DMUS_EVENTHEADER** structure, used only by applications that are directly sequencing events to the synthesizer.
- Dmksctrl.h: declarations for the <u>IKsControl</u> interface used for port property sets. You do not need this file if you have included Ksproxy.h and Ks.h.
- Dmplugin.h: 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.

You must also ensure that your application has access to the GUIDs used by DirectMusic. Define INITGUID before all other preprocessor directives, or link to Dxguid.lib.

DirectMusic 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 appropriate libraries.

Debugging DirectMusic Projects

The DirectMusic dynamic-link libraries (DLLs) installed with the debug version of the DirectX software development kit (SDK) generate information in the debug output window as the application is running. These DLLs are available if you installed the debug version of the DirectX SDK. They can be dynamically selected through the DirectX property sheet in Control Panel by choosing **Use Debug Version of 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 0 through 5, where 0 produces no debugging information and 5 the most detailed information. For most purposes it is unnecessary to set the level higher than 2; output at higher levels does not pertain to errors or warnings but is purely informational.

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 page of the DirectX property sheet in Control Panel. The slider sets the same value for all DLLs.

^{© 2004} Microsoft Corporation. All rights reserved.

First Steps in DirectMusic Programming

This topic gives an overview of the basic steps required for setting up a DirectMusic performance and playing a file. For details, see <u>Using DirectMusic</u>. For sample code, see <u>Tutorial 1: Playing Audio Files</u>.

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

- 1. Initialize COM. There are no helper functions for creating DirectMusic objects. To initialize COM, you must call **CoInitializeEx**.
- Create and initialize the performance. Most applications have a single performance object. Create the performance by calling **CoCreateInstance**, obtaining the <u>IDirectMusicPerformance8</u> interface. Then call <u>IDirectMusicPerformance8::InitAudio</u>. This method can set up a default audiopath.
- 3. Create the loader. Using **CoCreateInstance**, create a loader object and obtain an **IDirectMusicLoader8** interface. You need to do this only once, and normally you should keep the same loader object for the life of the application.
- Load a segment. Call <u>IDirectMusicLoader8::SetSearchDirectory</u> so the loader can find the data files. Then call <u>IDirectMusicLoader8::GetObject</u> to load a segment from a file or resource and obtain its <u>IDirectMusicSegment8</u> interface.
- 5. Download the band. Download <u>DLS</u> instrument data to the synthesizer so that notes can be synthesized. WAV files must also be downloaded. The simplest way to download all instruments and waves is by calling <u>IDirectMusicSegment8::Download</u>.
- 6. Play the segment. Pass the segment pointer to **IDirectMusicPerformance8::PlaySegmentEx**.

Using DirectMusic

This section is a guide to using the Microsoft® DirectMusic® application programming interface (API) in application development.

Information is presented in the following topics:

- Loading Audio Data
- <u>Playing Sounds</u>
- <u>Performance Parameters</u>
- <u>Using Audiopaths</u>
- <u>Using 3-D Sound in DirectMusic</u>
- <u>Using Effects in DirectMusic</u>
- Buffer Chains
- <u>Using Compositional Elements</u>
- <u>Using Audio Scripts</u>
- Capturing MIDI

For a more general overview, see the following topics:

- Elements of a DirectMusic Application
- First Steps in DirectMusic Programming
- **Overview of Audio Data Flow**

For information on advanced features used mainly by specialized applications, see <u>Advanced Topics in DirectMusic</u>.

Loading Audio Data

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.

Note Do not load data from untrusted sources. Loading DirectMusic data files causes objects to be constructed, with the possibility that excessive demand on resources will lead to degradation of performance or system failure.

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.

When objects are loaded from a memory location or a stream, the application should not touch the data until the loader is released. Because of caching and other internal mechanisms, the loader might try to access the data at a later time. To load new data, always allocate a new buffer or create a new stream.

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 have to use these interfaces directly.

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

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

• Setting the Loader's Search Directory

- <u>Scanning a Directory for Objects</u>
- Enumerating Objects
- Loading an Object from a File
- Loading an Object from a Resource or Memory Address
- <u>Containers</u>
- <u>Getting Object Descriptors</u>
- <u>Cache Management</u>
- Garbage Collection
- <u>Setting Objects</u>

See Also

• Custom Loading

Setting the Loader's Search Directory

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 and IDirectMusicLoader8::EnumObject methods. This default path can apply to all objects, or only to objects of a certain class.

The behavior of **IDirectMusicLoader8::LoadObjectFromFile** is somewhat different. See the Remarks for that method.

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

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

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

Scanning a Directory for Objects

The **IDirectMusicLoader8::ScanDirectory** method scans the current search directory for objects of a specified 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 **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 reside.

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. However, implementing a cache file does no harm.

For an example, see <u>Enumerating Objects</u>.

Enumerating Objects

Use the **IDirectMusicLoader8::EnumObject** method to iterate through all objects of a specified 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 <u>style</u> objects in the current search directory. The loop continues until there are no more objects of that class to enumerate.

```
void ListStyles(IDirectMusicLoader *pILoader)
{
  if (pILoader)
  {
    HRESULT hr = pILoader->SetSearchDirectory(
        CLSID DirectMusicStyle,
        L"c:\\mymusic",
        TRUE);
    if (SUCCEEDED(hr))
    Ł
      hr = pILoader->ScanDirectory(
          CLSID DirectMusicStyle,
          L"sty",*
          L"stylecache");
      if (hr == S_OK) // Only if files were found.
      {
        DWORD dwIndex;
        DMUS_OBJECTDESC objDesc;
        objDesc.dwSize = sizeof(DMUS_OBJECTDESC);
        for (dwIndex = 0; ;dwIndex++)
        {
          if (S_OK ==(pILoader->EnumObject(
              CLSID DirectMusicStyle,
              dwIndex, &objDesc)))
          {
             // Do something with information from objDesc.
```

```
}
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 index number that is not valid.

Loading an Object from a File

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

```
IDirectMusicLoader8* g_pLoader;
CoInitializeEx(NULL, 0);
HRESULT hr = CoCreateInstance(
    CLSID_DirectMusicLoader,
    NULL,
    CLSCTX_INPROC,
    IID_IDirectMusicLoader8,
    (void**)&g_pLoader);
if (FAILED(hr))
{
    ErrorExit(hr); // Add error-handling code.
}
```

You can then load an object from a 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 example code loads four segments from a directory previously set by **IDirectMusicLoader8::SetSearchDirectory**:

```
IDirectMusicSegment8 * g_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 = g_pLoader->LoadObjectFromFile(
    CLSID_DirectMusicSegment,
    IID_IDirectMusicSegment8,
```

```
wszNames[dwIndex],
   (void**) &g_pSegments[dwIndex]);
if (FAILED(hr))
   {
    ErrorExit(hr); // Add error-handling code.
   }
}
```

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

```
HRESULT LoadStyle(IDirectMusicStyle8 **ppStyle, IDirectMusicLoader8
Ł
  if (pLoader)
  {
    DMUS OBJECTDESC objDesc;
    // Start by initializing objDesc with the file name and
    // class GUID for the style object.
    wcsncpy(objDesc.wszFileName, L"c:\\mymusic\\polka.sty",
        sizeof(objDesc.wszName) - 1);
    objDesc.wszFileName[sizeof(objDesc.wszFileName) - 1] = 0;
    objDesc.guidClass = CLSID_DirectMusicStyle;
    objDesc.dwSize = sizeof (DMUS_OBJECTDESC);
    objDesc.dwValidData = DMUS_OBJ_CLASS |
        DMUS OBJ FILENAME
        DMUS_OBJ_FULLPATH;
    return pLoader->GetObject(&objDesc, IID_IDirectMusicStyle8,
        (void **) ppStyle);
  }
  else return E_INVALIDARG;
}
```

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 **wszName**, **guidObject**, or **wszFileName** in 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 <u>Scanning a Directory for Objects</u>.

See Also

• Loading an Object from a Resource or Memory Address.

Loading an Object from a Resource or Memory Address

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

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.

After **IDirectMusicLoader8::GetObject** has been called on an object in memory, the address and size of the memory buffer are privately cached by the loader. If you then release the buffer, a subsequent memory allocation might use the same address, and when another object is loaded from that address, the cached memory size will be used, possibly resulting in an incorrect number of bytes being loaded. To prevent this from happening, after loading the first object call **IDirectMusicLoader8::SetObject** with the same **DMUS_OBJECTDESC** descriptor, but with NULL in **pbMemData**. For this to work, the object descriptor must contain a name or GUID when passed to both **GetObject** and **SetObject**.

Note It is usually best not to release or change the contents of memory from which an object has been loaded, because it is difficult to be sure that the loader will not need to access the data again. In most cases, you should not release or reuse memory until after the loader is released. Also, do not load objects from data passed on the stack.

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

```
HRSRC hFound = FindResource(hMod, MAKEINTRESOURCE(ResourceID), RT
HGLOBAL hRes = LoadResource(hMod, hFound);
if (NULL == hRes) return E_FAIL;
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);
if (pLoader && ppSeg)
{
  hr = pLoader->GetObject(
      &objDesc, IID_IDirectMusicSegment8,
      (void**) ppSeg);
  return hr;
}
else return E INVALIDARG;
```

Objects referenced by other objects must be loaded first. For example, if you load a segment that contains a reference to a <u>style</u>, 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.

See Also

}

<u>Cache Management</u>

Containers

Containers are objects representing files that contain various objects. A container file might hold all the data necessary for a performance, including segments, <u>styles</u>, and <u>DLS</u> 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 example 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.

```
HRESULT LoadSegmentFromContainer (
    IDirectMusicLoader8* pLoader,
    WCHAR* wszFileName,
    WCHAR* wszSegmentName,
    IDirectMusicSegment** ppSeg)
{
    DMUS_OBJECTDESC objDesc;
    IDirectMusicContainer8* pContainer = NULL;
// Load the container.
    HRESULT hr = pLoader->LoadObjectFromFile(
        CLSID_DirectMusicContainer,
        IID_IDirectMusicContainer8,
        wszFileName,
        (void**)&pContainer);
    if (FAILED(hr)) return hr;
```

// 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**) ppSeg);
// 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();
  }
  return hr;
}
```

Getting Object Descriptors

After you have loaded an object, you can use its **<u>IDirectMusicObject8</u>** interface to retrieve information about it in a **<u>DMUS_OBJECTDESC</u>** structure.

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

```
void GetStyleName(IDirectMusicStyle8* pStyle)
{
  IDirectMusicObject8 *pIObject;
  DMUS_OBJECTDESC objDesc;
  if (SUCCEEDED(pStyle->QueryInterface(IID_IDirectMusicObject8,
      (void **) &pIObject)))
  {
    if (SUCCEEDED(pIObject->GetDescriptor(&objDesc)))
    ſ
      if (objDesc.dwValidData & DMUS_OBJ_NAME)
      {
        // Do something with objDesc.wszName,
        // which now contains the name of the style.
      }
    }
    pIObject->Release();
  }
}
```

Cache Management

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 clearing 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 <u>style</u>. 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 relies on the loader to keep the General MIDI <u>DLS</u> 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.

Note Regardless of whether caching is enabled, the loader keeps a private cache of object descriptors. When you load an object from a stream, a reference to the **IStream** is cached with the descriptor, and if that object is subsequently reloaded, it will be loaded from the same stream. When you load an object from a memory location, the address and buffer size are cached, and if the address is subsequently reused for another object, the incorrect number of bytes might be loaded. To ensure that the private cache of object descriptors is cleared, you can do one of the following:

- If caching is enabled, call **ClearCache**.
- If caching is not enabled, either destroy the loader and create a new one, or call **IDirectMusicLoader8::SetSearchDirectory** with the *fClear* parameter set to TRUE.

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.

See Also

- Garbage Collection
- Loading an Object from a Resource or Memory Address.

Garbage Collection

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, even if they are not being used 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. Objects only referenced by other objects that no longer exist are released. **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 must 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 the script track of a segment contains a reference to a script object, and this script object 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 the application 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 of the script object 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.

In the following example function, assume that the loaded script contains a reference to a segment. After calling a routine in the script, the function removes the script object from the cache and then calls **CollectGarbage**, which releases the referenced segment. If the segment contains a circular reference to the script, this is released so that the script can be destroyed, in turn releasing the final reference to the segment and allowing the segment to be destroyed.

```
void CallWhistle(IDirectMusicLoader8* pLoader, IDirectMusicPerforman
{
    IDirectMusicScript8 *pScript;
    WCHAR wszScript[MAX_PATH] = L"soundfx.spt";
    pLoader->LoadObjectFromFile(CLSID_DirectMusicScript,
```

See Also

<u>Cache Management</u>

Setting Objects

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 **<u>IDirectMusicLoader8::SetObject</u>** method takes as a parameter a **<u>DMUS_OBJECTDESC</u>** 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 <u>Loading an Object from a File</u> and <u>Loading an Object from a Resource or Memory Address</u>.
- 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:

```
HRESULT SetObjectFromResource(const GUID* guid, int ID,
    char* type, WCHAR* name, ÌDirectMusicLoader8* pLoader,
    HINSTANCE hInstance)
{
  HRSRC hResource = NULL;
  HGLOBAL hData = NULL;
  hResource = FindResource(hInstance, MAKEINTRESOURCE(ID), type);
  if (hResource != NULL)
  {
    hData = LoadResource(hInstance, hResource);
    if (hData != NULL)
      DMUS_OBJECTDESC objDesc;
      if(pLoader && (hResource != NULL) && (hData != NULL))
      Ł
        ZeroMemory(&objDesc,sizeof(objDesc));
        objDesc.pbMemData = (BYTE*) LockResource(hData);
```

```
objDesc.llMemLength = SizeofResource(hInstance, hResource);
        objDesc.guidClass = (*guid);
        objDesc.dwSize = sizeof(objDesc);
        objDesc.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_MEMORY;
        if (name)
        {
          wcsncpy(objDesc.wszName, name, sizeof(objDesc.wszName) - 1
          objDesc.wszName[sizeof(objDesc.wszName) - 1] = 0;
          objDesc.dwValidData |= DMUS_OBJ_NAME;
        }
        return pLoader->SetObject(&objDesc);
      }
    }
  }
  return E_FAIL;
}
```

The example 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:

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

Playing Sounds

This section describes basic techniques for playing sounds and coordinating the different elements of a performance.

The following topics are discussed:

- <u>Creating the Performance</u>
- <u>Using Segments</u>
- Changing the Pitch of Waveforms
- <u>Using Bands</u>
- <u>Timing</u>
- Notification and Event Handling
- <u>Troubleshooting Playback</u>

Creating the Performance

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 example function creates a performance and returns a pointer to the **IDirectMusicPerformance8** interface:

After the performance is created, it must be initialized. If your application is using audiopaths, you must call the **IDirectMusicPerformance8::InitAudio** method. Applications using the earlier channel-to-port mapping model must call **IDirectMusicPerformance8::Init** instead.

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 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 device 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 playing waveforms directly into DirectSound secondary 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 have to do this.

The following example function 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. The *hWnd* parameter is the application window handle.

```
BOOLEAN Init(IDirectMusicPerformance8* pPerf, HWND hWnd)
{
    if (FAILED(pPerf->InitAudio(NULL, NULL, hWnd,
        DMUS_APATH_SHARED_STEREOPLUSREVERB, 16,
        DMUS_AUDIOF_ALL, NULL)))
    {
        return FALSE;
    }
    else return TRUE;
}
```

Using Segments

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

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.

This section provides more information on segments in the following topics:

- <u>Creating Segment Objects</u>
- <u>Playing Segments</u>
- <u>Segment States</u>
- <u>Pausing Segments</u>
- <u>Control Segments</u>
- <u>Self-Controlling Segments</u>
- MIDI-Based Segments
- <u>WAV-Based Segments</u>

Creating Segment Objects

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

- Load a file or resource object that supports the **IDirectMusicSegment8** interface. Most segments are created this way.
- Get a motif from a <u>style</u> by using the <u>IDirectMusicStyle8::GetMotif</u> method.
- Use methods of the **IDirectMusicComposer8** interface to create a composition or transition at run time.
- Make a copy of an existing segment by using the **IDirectMusicSegment8::Clone** method.
- Use the **IDirectMusicBand8::CreateSegment** method. This method creates a special type of secondary segment that is used only for making band changes.
- 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 existing segments.
- Construct a segment from existing tracks. To do this, create a segment object by calling CoCreateInstance, and then add tracks by calling <u>IDirectMusicSegment8::InsertTrack</u>. This technique is not used by most applications.

See Also

- <u>DirectMusic Tracks</u>
- Loading Audio Data
- <u>Making Band Changes Programmatically</u>
- <u>Using Compositional Elements</u>
- <u>Using Templates</u>
- <u>Using Transitions</u>

Playing Segments

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, setting global parameters such as tempo. Secondary 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, three kinds of segments have 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, <u>groove levels</u>, and embellishments, used in conjunction with a style and <u>chordmap</u> to compose music at run time.

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

Only one primary segment at a time can be played. 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.

See Also

- <u>Segment Timing</u>
- Making Band Changes Programmatically
- <u>Using Templates</u>
- <u>Using Transitions</u>

Segment States

When you play a segment, parameters for that instance of the playing segment (such as the audiopath, start point, and number of repetitions) are stored in an object called a segment state, represented by an **IDirectMusicSegmentState8** interface.

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.

See Also

• <u>Retrieving Objects from an Audiopath</u>

Pausing Segments

To pause a segment, you must ascertain the current play position before stopping the segment. The following example function returns the current play position in music time.

```
MUSIC_TIME GetTimeOffset(const MUSIC_TIME mtNow,
                                                                      // From G
                             const MUSIC_TIME mtStartTime, // From G
const MUSIC_TIME mtStartPoint, // From G
const MUSIC_TIME mtLoopStart, // From G
const MUSIC_TIME mtLoopEnd
                             const MUSIC_TIME mtLoopEnd, // From G
const MUSIC_TIME mtLength, // From G
const DWORD dwLoopRepeats) // From G
{
    // Convert mtNow from absolute time to an offset
    // from when the segment started playing.
    LONGLONG lloffset = mtNow - (mtStartTime - mtStartPoint);
    // If mtLoopEnd is not zero, set llLoopEnd to mtLoopEnd;
    // otherwise use the segment length.
    LONGLONG llLoopEnd = mtLoopEnd ? mtLoopEnd : mtLength;
    LONGLONG llLoopStart = mtLoopStart;
    // Adjust offset to take looping into account.
    if ((dwLoopRepeats != 0) && (llLoopStart < llLoopEnd) && (llLo
    {
         if ((dwLoopRepeats != DMUS_SEG_REPEAT_INFINITE)
           && (llOffset > (llLoopStart + (llLoopEnd - llLoopStart) *
         {
              llOffset -= (llLoopEnd - llLoopStart) * dwLoopRepeats;
         }
         else if (ll0ffset > llLoopStart)
         {
              110ffset = llLoopStart + (llOffset - llLoopStart) % (llL
         }
    }
    110ffset = min(ll0ffset, LONG MAX); // LONG MAX is defined in L
    return long(ll0ffset);
```

}

To restart the segment at the correct position, pass the return value of the sample function to **IDirectMusicSegment8::SetStartPoint** before calling **IDirectMusicPerformance8::PlaySegmentEx**.

Note The *mtLength* parameter of the example function will normally be 1 for segments loaded from WAV files. In this case, before calling **SetStartPoint** you must use **IDirectMusicSegment8::SetLength** to set the length of the segment to at least 1 tick more than the current offset. For more information, see **IDirectMusicSegment8::SetStartPoint**.

Control Segments

The control segment is the source of any data that is shared across tracks by parameter calls.

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 chord, command, mute, and tempo 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.

See Also

- DMUS_SEGF_FLAGS
- <u>Self-Controlling Segments</u>
 <u>Track Configuration</u>

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 commands (such as groove levels) rather than the commands in 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.

Configuring a segment as self-controlling is usually done 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:

Flag	Effect
DMUS_TRACKCONFIG_OVERRIDE_ALL	The track should get parameters from this segment before control and primary segment tracks.
DMUS_TRACKCONFIG_OVERRIDE_PRIMARY	The track should get parameters from this segment before primary segment tracks.
DMUS_TRACKCONFIG_FALLBACK	The track should get parameters from this segment if the primary and control segments do not return the needed

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.

See Also

- <u>Control Segments</u>
- <u>Track Configuration</u>.

MIDI-Based Segments

A MIDI-based segment can be created at run time by loading a standard MIDI (.mid) file. In addition, MIDI-based segment (.sgt) files can be created in DirectMusic Producer, often by importing MIDI files to which the author might add tempo, key, and band changes, as well as loop points. Unlike a <u>style</u>-based segment, a MIDI-based segment has no <u>patterns</u> and no command track. Instead, it has a sequence track that contains MIDI notes and other commands.

WAV-Based Segments

DirectMusic can load and play WAV files, but greater flexibility is provided by DirectMusic Producer segments containing wave tracks. A segment can contain any number of wave tracks, and a wave track can contain multiple waveforms.

WAV-based DirectMusic Producer segments can be used to create sound effects and ambient audio that do not sound repetitive. Each part in a wave track can have up to 32 variations, and every waveform in the part is assigned to one or more of these variations. Each time the track plays, one of the variations is selected for each part, and only the waveforms assigned to that variation are heard. Because different parts can play different variations, a two-part wave track could play in up to 1024 different ways. For an example of the use of waveform variations, see the <u>Audio Scripts Sample</u>.

Waveform variations are authored into segments. Applications do not control the selection of variations.

Another advantage of WAV-based segments over WAV files is that the waveforms can be in compressed format. The DirectMusic loader can load compressed waveform audio files in any format supported by the audio compression manager (ACM).

Segments loaded from WAV 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 waveforms are not synthesized in the same sense as musical notes, they do pass through the synthesizer and can be manipulated by MIDI controllers.

Waveforms are analogous to band instruments and must be downloaded to the synthesizer before being played. Waveforms are downloaded when the segment's bands are downloaded.

Waveforms can be either static or streaming. Static waveforms are loaded into synthesizer memory all at once. Streaming waveforms are loaded piece by piece as they play. Waveforms saved as DirectMusic Producer files are designated as static or streaming by the author, who also sets the readahead time; that is, the maximum amount of data that is copied into memory at one time. Waveforms from standard WAV files are streamed if longer than 5000 milliseconds, with a readahead of 500 milliseconds.

Playing WAV Sounds in Music Time

Because a WAV sound has no tempo, the loader cannot calculate the music-time length of a segment loaded from a WAV file or resource, and it always sets the length to 1 tick. In consequence, you cannot cue another segment to play immediately after a WAV segment by using the DMUS_SEGF_QUEUE flag. A further limitation is that you must play the WAV segment from the beginning, because **IDirectMusicSegment8::SetStartPoint** fails with any parameter greater than the known music-time length of the segment.

To overcome these limitations, you must set the length of the segment to a music-time value equivalent to the clock-time length of the sound. The length of the sound can be calculated from the WAV chunk headers. The following sample function sets the length of a WAV segment whose length is known:

See Also

- <u>Clock Time and Music Time</u>
- <u>IDirectMusicSegment8::SetLength</u>

Changing the Pitch of Waveforms

Several techniques can be used to change the pitch of a waveform programmatically.

If you are playing the waveform as a segment, you can obtain the DirectSound buffer through which it is playing and use the **IDirectSoundBuffer8::SetFrequency** method to change the pitch. Because a pitch change affects all sounds playing through a buffer, you should play the

sound on its own audiopath, or its own mix group within an audiopath. Different buffers within a mix group cannot have different frequencies. (For information on mix groups, see the Help for DirectMusic Producer.)

This technique does not work well with looping segments. Because the performance does not take into account the actual time it takes for the sound to finish playing in the buffer, looping continues to happen at intervals based on the normal length of the sound. When the pitch is lower than the original pitch of the sound, the whole sound does not play before it loops; when the pitch is higher than the original, intervals of silence occur between loops. If the entire sound is looping, you can adjust the intervals by using

IDirectMusicSegment8::SetLength before playing the segment; however, calling this method while the segment is playing will have unpredictable results.

Another technique is to set a MIDI pitch bend on the performance channel on which the sound is playing. To do so, send a **DMUS_MIDI_PMSG** message, specifying 0xE0 as the status byte. The lower seven bits of the two data bytes are combined as a 14-bit value, where 0x2000 specifies no pitch bend, lower values represent a lower pitch, and higher values represent a higher pitch. For more information, see the MIDI specification.

Perhaps the most effective way to control the pitch is to play the waveform as a DLS instrument created in DirectMusic Producer. You can use the same WAV sound as the sample for all regions (note ranges), or assign different sounds to different regions for more realistic effects. You can also create articulations such as volume envelopes to make the sound start and stop in a more natural way. To play the sound at different pitches in your application, start and stop different notes by sending DMUS_NOTE_PMSG messages.

See Also

- <u>Changing the Tempo</u>
- <u>Retrieving Objects from an Audiopath</u>
 <u>Using DirectMusic Messages</u>

Using Bands

A band is a collection of settings for <u>performance channels</u>. Every part in a segment or <u>style</u> plays on a single performance channel, and the most recently applied band determines how sound is synthesized on that channel.

Bands can contain the following information for each channel:

- Instrument, identified by a MIDI patch number and <u>bank select</u>. This setting does not affect WAV parts playing on the channel.
- <u>DLS</u> collection from which to load the instrument. By default, the DLS collection is the standard General MIDI collection.
- Volume.
- Pan.
- Transposition. If this value is not zero, music notes on the channel are automatically transposed for the instrument.

A band does not necessarily contain settings for every performance channel in use, and it does not have to contain every possible setting for channels it does affect. Settings not explicitly changed by a band remain as they were before the band was played. For example, if the application plays a band that does not contain any settings for channel 1, any parts on that channel continue playing on the last instrument assigned to it, and with the same volume, pan, and transposition settings. The band could change a single setting, such as the volume, without affecting any of the other settings currently in effect.

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.

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. When you have obtained an **IDirectMusicBand8** interface, you have access to that band and can substitute it for the default band.

See Also

- Downloading and Unloading Bands
- <u>Making Band Changes Programmatically</u>
- 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 <u>performance channels</u> and downloads the <u>DLS</u> 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 <u>Making Band Changes Programmatically</u>.

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

- <u>Automatically Downloading Bands</u>
- <u>Manually Downloading Bands</u>
- Patch Collisions
- <u>Unloading Bands</u>

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 is time-consuming 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.

You can turn on automatic downloading of bands in one of the following ways:

- Call the **IDirectMusicPerformance8::SetGlobalParam** method for the GUID_PerfAutoDownload parameter.
- Enable automatic downloading for a single segment by calling the **IDirectMusicSegment8::SetParam** method for the <u>GUID_Enable_Auto_Download</u> parameter.

In the following example function, the global parameter for the performance is set to enable automatic downloading of bands in all segments:

```
HRESULT TurnOnDownload(IDirectMusicPerformance8* pPerf)
{
   BOOL fAuto = TRUE;
   HRESULT hr = pPerf->SetGlobalParam(
        GUID_PerfAutoDownload, &fAuto, sizeof(BOOL));
```

```
return hr;
}
```

See Also

- <u>Setting and Retrieving Global Parameters</u>
- <u>Setting and Retrieving Track Parameters</u>
- <u>Unloading Bands</u>

Manually Downloading Bands

You can manually download a band in one of the following ways:

- Call **<u>IDirectMusicSegment8::Download</u>** to download the bands and waveforms 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 <u>IDirectMusicSegment8::SetParam</u> method with the <u>GUID_Download</u> or <u>GUID_DownloadToAudioPath</u> parameter to download the band in the segment's first band track. You can also use <u>IDirectMusicPerformance8::SetParam</u> to set this parameter on the primary segment, or <u>IDirectMusicTrack8::SetParamEx</u> to set it directly on a band track. For more information, see <u>Setting and Retrieving Track</u> <u>Parameters</u>.

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

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 example function loads a band from a file and downloads it to the performance:

```
HRESULT DownloadBand(
   IDirectMusicLoader8 *pLoader,
   IDirectMusicPerformance8 *pPerf,
   WCHAR *pwszFile)
{
   IDirectMusicBand8* pBand;
```

Patch Collisions

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

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 <u>performance channel</u> 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

Bands take up memory and 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.

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

See Also

• <u>Setting and Retrieving Track Parameters</u>

Making Band Changes Programmatically

In most cases, the band track in a loaded segment performs all necessary program changes. However, you can also do so manually if you have a band object. You must create a secondary segment by using **IDirectMusicBand8::CreateSegment**, 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 example function creates a segment from a band and plays it on the next measure boundary. It is presumed that the instruments have already been downloaded or that automatic downloading has been enabled.

See Also

```
• Patch Collisions
```

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 than the beat boundary. If you make a band change at the beat boundary, the note might play with the incorrect instrument.

To prevent this problem, an application should cue the band segment early. Suppose, for example, that you have a style-based 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 example code ensures that the band segment is played 31 ticks before the style segment, so that all instruments are in place before any note is played:

```
HRESULT CueSegmentAfterBand(IDirectMusicPerformance8* pPerf,
                            IDirectMusicSegment8* pBandSeg,
                            IDirectMusicSegment8* pStyleSeg)
{
  REFERENCE_TIME rtResolved;
  MUSIC TIME mtResolved;
  HRESULT hr;
  hr = pPerf->GetResolvedTime(0, &rtResolved, DMUS_TIME_RESOLVE_MEAS
  if (SUCCEEDED(hr))
  {
    hr = pPerf->ReferenceToMusicTime(rtResolved, &mtResolved);
   if (SUCCEEDED(hr))
   {
      mtResolved -= 31;
      hr = pPerf->PlaySegment(pBandSeg, 0, mtResolved, NULL);
      if (SUCCEEDED(hr))
      {
        pPerf->PlaySegment(pStyleSeg, DMUS_TIME_RESOLVE_MEASURE, 0,
      }
    }
  }
  return hr;
}
```

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 <u>DLS</u> collection when playing a MIDI file.

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 objDesc;
  mbstowcs(objDesc.wszFileName,pszFileName,DMUS_MAX_FILENAME);
  objDesc.dwSize = sizeof(DMUS_OBJECTDESC);
  objDesc.guidClass = CLSID_DirectMusicCollection;
  objDesc.dwValidData = DMUS_OBJ_CLASS
      | DMUS OBJ FILENAME
      | DMUS_OBJ_FULLPATH;
  hr = pILoader->GetObject(&objDesc,
    IID IDirectMusicCollection8,
    (void **) ppICollection);
  return hr;
}
```

Next, you must associate the DLS data with the segment by calling **IDirectMusicSegment8::SetParam**, as shown in the following example:

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

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.

See Also

• <u>Using Instrument Collections</u>.

Timing

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

- Master Clock
- <u>Clock Time and Music Time</u>
- <u>Changing the Tempo</u>
- <u>Prepare Time</u>
- Latency and Bumper Time
- <u>Reducing Latency</u>
- <u>Segment Timing</u>

Master Clock

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. After you have obtained the GUID of the device that you want to use as the master clock, pass it to the **IDirectMusic8::SetMasterClock** method.

Clock Time and Music Time

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.

For example, suppose 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.

See Also

• Playing WAV Sounds in Music Time

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 <u>Clock Time and Music Time</u>.) The tempo track of the control segment usually manages the tempo, but it is also possible for an application to set the tempo dynamically.

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

The following example function sends a message to change the tempo, after disabling the tempo track so that it does not override the new tempo.

```
HRESULT ChangeTempoByMessage(IDirectMusicPerformance8* pPerformance,
                    IDirectMusicSegment8* pSegment,
                    double dblNewTempo)
{
   DMUS_TEMPO_PMSG* pTempoMsg;
   HRESULT hr;
   if (SUCCEEDED(hr = pSegment->SetParam(GUID_DisableTempo,
          0xFFFF, 0, 0, NULL)))
   {
    if (SUCCEEDED(hr = pPerformance->AllocPMsg(
        sizeof(DMUS_TEMPO_PMSG), (DMUS_PMSG**)&pTempoMsg)))
    {
      // Cue the tempo event.
      ZeroMemory(pTempoMsg, sizeof(DMUS_TEMP0_PMSG));
      pTempoMsq->dwSize = sizeof(DMUS_TEMPO_PMSG);
      pTempoMsg->dblTempo = dblNewTempo;
      pTempoMsg->dwFlags = DMUS_PMSGF_REFTIME;
      pTempoMsg->dwType = DMUS_PMSGT_TEMP0;
      pPerformance->SendPMsg((DMUS_PMSG*)pTempoMsg);
    }
  }
  return hr;
}
```

If the performance has more than one audiopath, the message should be stamped for delivery to the correct audiopath. For more information, see <u>Application-</u>

Created Messages.

The following example shows how to change the tempo parameter.

```
HRESULT ChangeTempoParameter(IDirectMusicSegment8* pSegment, double
{
    DMUS_TEMP0_PARAM Tempo;
    Tempo.dblTempo = dblNewTempo;
    HRESULT hr = pSegment->SetParam(GUID_TempoParam, 0xFFFF, 0, 0, &Te
    return hr;
}
```

Note <u>DMUS_TEMPO_PARAM</u> is declared in Dmusicf.h, which is not automatically included when you include Dmusici.h.

You can also change the master tempo, which adjusts the tempo set by any control segment.

See Also

- Setting and Retrieving Global Parameters
- <u>Setting and Retrieving Track Parameters</u>

Prepare Time

As a segment is played, the performance makes repeated calls to the segment's tracks, causing them to generate messages for a specified time range, which is a 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 time.

Each time it calls on a track to play messages, the performance calculates the end time for that call by adding the prepare time to the current time. For example, if the current time is 10,000 milliseconds (or the equivalent in reference time units) and the prepare time is the default 1000 milliseconds, the end time is at 11,000. The result is that all new messages that are to be played up to time 11,000 must be prepared and placed in the queue.

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**.

If you set a start time of "now" for the segment without specifying DMUS_SEGF_AFTERPREPARETIME, the performance invalidates any messages currently in the queue. Any tracks that are still valid at this point (for example, tracks of secondary segments, which continue to play) 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 are in the queue and the current time is 9,000, a segment cued to play immediately, but flagged DMUS_SEGF_AFTERPREPARETIME, starts playing just after the 10,000 mark.

See Also

- <u>IDirectMusicPerformance8::GetPrepareTime</u>
- <u>IDirectMusicPerformance8::SetPrepareTime</u>
- Latency and Bumper Time
- <u>Segment Timing</u>.

Latency and Bumper Time

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 waveform to play. The **IDirectMusicPerformance8::GetLatencyTime** method retrieves the current time plus the latency for the performance as a whole. The latency is based on the largest value returned by any port's latency clock.

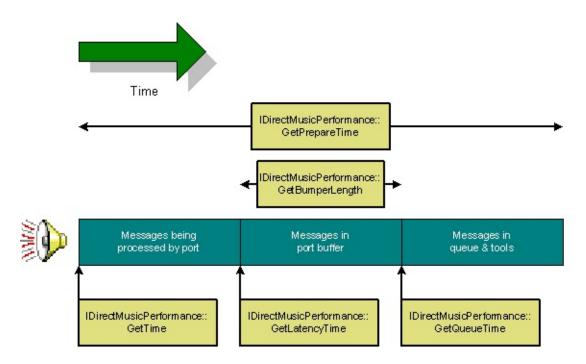
The bumper is an extra amount of time allotted for code to run between the time that an event is put into the port buffer and the time that the port starts to process it. By default, the bumper length is 50 milliseconds.

The following example shows how latency time and bumper time are combined. Suppose an event is cued to play at 10,000 milliseconds. 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.

After 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.

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 moment.



See Also

- <u>Reducing Latency</u>
- <u>IDirectMusicPerformance8::GetBumperLength</u>.
- IDirectMusicPerformance8::SetBumperLength

Reducing Latency

The latency in DirectMusic consists of two major components, systemdependent latency and latency padding.

System-dependent latency is calculated dynamically by approximating the minimum amount of write-ahead data required for the given system configuration. This behavior is automatic and not controllable by the application. The absolute minimum system-dependent latency is 4 milliseconds.

Latency padding ranges from 0 to 100 milliseconds, and is configurable by the application. This value is added to the system-dependent latency and may be used to mitigate glitching problems. To ensure maximum compatibility with older systems, latency padding is set to 55 milliseconds by default. Most applications do not need to increase the latency padding.

Decreasing the latency padding allows applications to take advantage of the lowlatency capabilities of modern hardware. However, doing so comes at the risk of glitching on some systems. Only applications that truly need the lowest latency possible should reduce the latency padding value.

Changing the latency padding value is accomplished by setting the GUID_DMUS_PROP_WriteLatency property on the port. The following example code demonstrates the use of this property.

```
HRESULT SetLatency (IDirectMusicPort8 *pDMPort, DWORD dwLatency)
{
    IKsControl* pKSControl;
    HRESULT hr;
    // Query for IKsControl. All ports that support properties provi
    hr = pDMPort->QueryInterface(IID_IKsControl, (void**)&pKSControl
    if (SUCCEEDED(hr)) {
        KSPROPERTY KSProperty;
        ULONG       ulDummy;
        ZeroMemory(&KSProperty, sizeof(KSProperty));
        KSProperty.Set = GUID_DMUS_PROP_WriteLatency;
        KSProperty.Flags = KSPROPERTY_TYPE_SET;
        hr = pKSControl->KsProperty(&KSProperty, sizeof(KSProperty);
```

```
(LPVOID)&dwLatency, sizeof(dwLatency), &ulDummy);
    pKSControl->Release();
  }
  return hr;
}
```

To attain the absolute minimum latency, applications must also reduce the wakeup interval of the DirectMusic realtime thread. This is achieved by setting the GUID_DMUS_PROP_WritePeriod property on the port.

See Also

- **KSPROPERTY**
- <u>Property Sets for DirectMusic Ports</u>

Segment Timing

When you cue a segment to play, you have a great deal of control over when the segment starts, what point in the segment is heard first, how the segment is aligned rhythmically with currently playing music, and whether any part of the segment plays more than once.

The following terms are used throughout this section to clarify the relationship between times within segments and times within the performance.

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 than the start point 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 segment's start point is cued. In the DirectMusic API, this time is sometimes called start time.

Resolved time

A specified time (such as the play time) adjusted to a specified boundary. For example, the resolved time could be the time of the next beat after the specified time.

Start marker

A marker indicating a valid segment start time in a segment. The marker can be

an enter switch point in the marker track, or a variation switch point in a <u>pattern</u>.

Play marker

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

Note Start markers and play markers are placed in a segment in DirectMusic Producer and cannot be changed by the application. They can, however, be retrieved by using the <u>GUID_Valid_Start_Time</u> and <u>GUID_Play_Marker</u> 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.

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 can 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.

Repeating and Looping

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.

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, DMUS_SEGF_SEGMENTEND, 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.

Note Combining DMUS_SEGF_ALIGN with DMUS_SEGF_SEGMENTEND causes the beginning of the cued segment to be aligned with the beginning of the current 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:

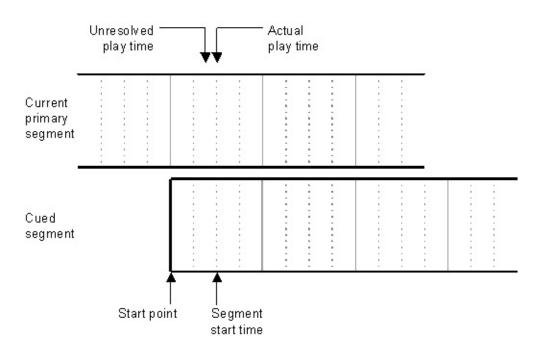
Flag	Effect
DMUS_SEGF_VALID_START_BEAT	Puts the segment start time on the next beat.
DMUS_SEGF_VALID_START_GRID	Puts the segment start time on the next grid.
DMUS_SEGF_VALID_START_MEASURE	Puts the segment start time on the next bar line.
	Puts the segment start time at

• 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 <u>Clock</u> <u>Time and Music Time</u>.

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 and 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.

© 2004 Microsoft Corporation. All rights reserved.

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 occurs.

Note Performance notifications should not be confused with DirectSound buffer notifications, which are not used by DirectMusic applications.

To specify what types of events you want to be notified of, call the **IDirectMusicPerformance8::AddNotificationType** method once for each desired type of event.

The following example function causes DirectMusic to set segment-related events. The specific type of event, such as a segment start or a segment end, is derived later from the notification message.

```
HRESULT SetSegmentNotification(IDirectMusicPerformance8* pPerformanc
{
   GUID guid = GUID_NOTIFICATION_SEGMENT;
   return 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 call the **IDirectMusicTrack8::AddNotificationType** method directly.

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 **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:
```

```
HRESULT hr = g_pPerf->SetNotificationHandle(g_hNotify, 0);
```

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

```
void WaitForEvent(LPVOID lpv, HANDLE hNotify, IDirectMusicPerformanc
{
    DWORD dwResult;
    DMUS_NOTIFICATION_PMSG* pPmsg;
    while (TRUE)
    {
        dwResult = WaitForSingleObject(hNotify, 100);
        while (S_OK == pPerformance->GetNotificationPMsg(&pPmsg))
        {
            // Check notification type and do something in response.
            // Then free the message.
            pPerformance->FreePMsg((DMUS_PMSG*)pPmsg);
        }
    }
}
```

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.

^{© 2004} Microsoft Corporation. All rights reserved.

Troubleshooting Playback

The following are common causes for sounds not playing correctly:

 Instruments have not been downloaded. You must download all instruments and waveforms used by segments, even the default Gm.dls instrument collection. To ensure that all instruments and waveforms are downloaded, you can turn on automatic downloading; however, this is not recommended. A better way is to download each segment when it is loaded, by using IDirectMusicSegment8::Download. For more information, see Downloading and Unloading Bands.

There is one case in which automatic downloading might be required: when a segment trigger track cues other segments that use different instruments or waveforms. Because your application doesn't obtain an interface to the triggered segments, it cannot download them manually.

- The loader cannot find needed files. Make sure the loader is searching in the correct folder by calling **IDirectMusicLoader8::SetSearchDirectory**.
- The loader cannot find objects that are used by other objects. Content authored in DirectMusic Producer can contain references to content in other files. For example, a script track in a segment might contain calls to play other segments. If the loader cannot find the referenced segments, it cannot play them. To ensure that the loader can identify and load all referenced objects, call **IDirectMusicLoader8::ScanDirectory**. For more information, see <u>Scanning a Directory for Objects</u>.

Performance Parameters

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 <u>Control Segments</u>.
- **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.

See Also

- <u>Setting and Retrieving Track Parameters</u>
- <u>Disabling and Enabling Track Parameters</u>
- <u>Setting and Retrieving Global Parameters</u>

Setting and Retrieving Track Parameters

The following methods are used for setting and retrieving track parameters:

- IDirectMusicPerformance8::GetParam
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- IDirectMusicSegment8::SetParam
- <u>IDirectMusicTrack8::GetParam</u>
- <u>IDirectMusicTrack8::SetParam</u>
- IDirectMusicTrack8::GetParamEx
- <u>IDirectMusicTrack8::SetParamEx</u>

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 <u>Identifying the Track</u>.

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 <u>Standard Track Parameters</u>.

Some parameter changes might not appear to take effect immediately. For example, changing the <u>groove level</u> does not make a difference until the current <u>pattern</u> 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 <u>IDirectMusicPerformance8::Invalidate</u> method.

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

Identifying the Track

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 0xFFFFFFF 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:

```
HRESULT hr = pIDMSegment->SetParam(GUID_DisableTempo, 0xFFFFFFF,
DMUS_SEG_ALLTRACKS, 0, NULL);
```

There are times, however, when you need to specify 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. 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	
В	0x2	
С	0x1	
D	0x3	

Group 1 contains tracks A, C, and D, and group 2 contains tracks B and D. If you call **GetParam** 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 one bit in *dwGroupBits*, the parameter is retrieved from the *n*th track containing any of those bits, where *n* is the value in *dwIndex*.

See Also

• IDirectMusicSegment8::InsertTrack

Disabling and Enabling Track Parameters

By setting the <u>GUID_DisableTempo</u> and <u>GUID_DisableTimeSig</u> 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 <u>GUID_EnableTempo</u> or <u>GUID_EnableTimeSig</u> 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.

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

See Also

- <u>Control Segments</u>
- IDirectMusicTrack8::IsParamSupported
- Setting and Retrieving Track Parameters.

Setting and Retrieving Global Parameters

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 (<i>rguidType</i>)	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 <u>Downloading and Unloading</u> <u>Bands</u> .
GUID_PerfMasterGrooveLeve	l char	The master <u>groove level</u> 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

GUID_PerfMasterVolume	long	DMUS_MASTERTEMPO_MAX. 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. Setting this parameter is equivalent to calling IKsControl::KsProperty for the GUID_DMUS_PROP_Volume property set on every port in the performance.
-----------------------	------	--

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

Note All parameters have to be set before they can be retrieved. When a parameter is set, the performance allocates memory for the data in a linked list of items that are identified by GUID. If **SetGlobalParam** has never been called on the parameter, it does not appear in this linked list, and **GetGlobalParam** fails.

^{© 2004} Microsoft Corporation. All rights reserved.

Using Audiopaths

A DirectMusic performance typically 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 <u>DMOs</u>, 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 DirectMusic you may want to use multiple audiopaths and exercise more control over them.

Audiopaths use only the Microsoft software synthesizer. If you want your application to use another port or ports, you must initialize the performance by using **IDirectMusicPerformance8::Init**. For more information, see <u>Using DirectMusic Ports</u>.

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

- Creating Audiopaths
- Default Audiopath
- <u>Standard Audiopaths</u>
- Playing Sounds on Audiopaths
- Retrieving Objects from an Audiopath

Creating Audiopaths

Applications can set up audiopaths in the following ways:

- Create one or more standard audiopaths by using <u>IDirectMusicPerformance8::CreateStandardAudioPath</u>. See <u>Standard</u> <u>Audiopaths</u>.
- Create a default standard audiopath for the performance in the call to **IDirectMusicPerformance8::InitAudio**. See <u>Default Audiopath</u>.
- 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.
- Obtain an audiopath configuration from a file authored in DirectMusic Producer and pass the configuration object to IDirectMusicPerformance8::CreateAudioPath.

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 example code, see <u>Playing Sounds on Audiopaths</u>.

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

Default Audiopath

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**.

Standard Audiopaths

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

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

The audiopaths defined by DirectMusic manage the flow of synthesizer output through combinations of standard buffers, some of which have effect <u>DMOs</u> 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	Music	Yes
DW05_AFAIII_SHARED_STEREOFLOSREVERD	Reverb	Yes

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 be obtained from the buffer object. In addition, applications can add effects to buffers by using **IDirectSoundBuffer8::SetFX**, making other interfaces available. For more information on obtaining interfaces, see <u>Retrieving Objects from an Audiopath</u>.

Standard buffer	escription	Capabilities	Interfaces
		DSBCAPS_CTRL3D	

3-D Dry	Mono 3-D buffer	DSBCAPS_CTRLFREQUENCY DSBCAPS_CTRLFX DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS DSBCAPS_MUTE3DATMAXDISTANCE	IDirectSoun IDirectSoun IDirectSoun
Mono	Mono buffer with no effects	DSBCAPS_CTRLFREQUENCY DSBCAPS_CTRLFX DSBCAPS_CTRLPAN DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS	IDirectSoun
Music	Stereo buffer used with Reverb		IDirectSoun
Reverb	with music	DSBCAPS_CTRLFX DSBCAPS_CTRLPAN DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS	IDirectSoun IDirectSoun
Stereo	Stereo buffer with no effects	DSBCAPS_CTRLFREQUENCY, DSBCAPS_CTRLFX, DSBCAPS_CTRLPAN DSBCAPS_CTRLVOLUME DSBCAPS_GLOBALFOCUS	IDirectSoun

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

- <u>DMUS_APATH_DYNAMIC_3D</u>
- <u>DMUS_APATH_DYNAMIC_MONO</u>
- DMUS_APATH_DYNAMIC_STEREO
- <u>DMUS_APATH_SHARED_STEREOPLUSREVERB</u>

DMUS_APATH_DYNAMIC_3D

This standard audiopath sets up a nonshared buffer of type 3-D Dry.

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.

The buffer uses the DS3DALG_NO_VIRTUALIZATION algorithm for 3-D effects, and this property cannot be changed by the application. Other algorithms can be applied to custom buffers in audiopaths that have been authored in DirectMusic Producer. For information on 3-D algorithms, see DSBUFFERDESC in the DirectX documentation.

See Also

• **Standard Audiopaths**

DMUS_APATH_DYNAMIC_MONO

This standard audiopath sets up a nonshared buffer of type Mono that has no 3-D parameters or special effects.

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.

See Also

• <u>Standard Audiopaths</u>

DMUS_APATH_DYNAMIC_STEREO

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

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.

See Also

• <u>Standard Audiopaths</u>

DMUS_APATH_SHARED_STEREOPLUSREVERB

This standard audiopath 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.

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 example function retrieves an **IDirectSoundFXWavesReverb8** interface to the <u>DMO</u> in the Reverb buffer on a default DMUS_APATH_SHARED_STEREOPLUSREVERB audiopath:

```
HRESULT GetDMO(IDirectMusicPerformance8* pPerf,IDirectSoundFXWavesRe
{
    IDirectMusicAudioPath8 * pAudioPath;
    HRESULT hr;
    hr = pPerf->GetDefaultAudioPath(&pAudioPath);
    if (SUCCEEDED(hr))
    {
        HRESULT hr = pAudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,
            DMUS_PATH_BUFFER_DMO, 1,
            GUID_All_Objects, 0, IID_IDirectSoundFXWavesReverb8,
            (LPV0ID*) ppEffectDMO);
    }
    return hr;
}
```

See Also

• <u>Standard Audiopaths</u>

Playing Sounds on Audiopaths

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 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 <u>Automatically Downloading Bands</u>.

The following example function plays a segment on an embedded audiopath configuration if one is available, or on the default audiopath otherwise:

```
HRESULT PlayOnEmbedded(IDirectMusicPerformance8* pPerf, IDirectMusic
{
    IDirectMusicAudioPath8 * pPath = NULL;
    IUnknown *pConfig;
    HRESULT hr;
    if (pSeg)
```

```
{
    if (SUCCEEDED(hr = pSeg->GetAudioPathConfig(&pConfig)))
    {
        hr = pPerf->CreateAudioPath(pConfig, TRUE, &pPath);
        pConfig->Release();
    }
    hr = pPerf->PlaySegmentEx(pSeg, NULL, NULL, 0, 0, NULL, NULL, pP
    if (pPath)
    {
        pPath->Release();
        pPath = NULL;
    }
    }
    return hr;
}
```

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 <u>performance channels</u> playing on this audiopath.

Retrieving Objects from an Audiopath

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 <u>DMO</u> 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
DMUS_PATH_AUDIOPATH (*)	Ignored
DMUS_PATH_AUDIOPATH_GRAPH	Ignored
DMUS_PATH_AUDIOPATH_TOOL	Tool class ID or GUID_All_Objects to enumerate
DMUS_PATH_BUFFER	Ignored
DMUS_PATH_BUFFER_DMO	DMO class ID, such as GUID_DSFX_STANDARD_GARG

GUID_All_Objects to enumerate

DMUS_PATH_MIXIN_BUFFER_DMO or GUID_All_Objects to enumerate	[2]	
DMUS_PATH_PERFORMANCE Ignored		
DMUS_PATH_PERFORMANCE_GRAPH Ignored		
DMUS_PATH_PERFORMANCE_TOOL Tool class ID or GUID_All_Objects to enumerate		
DMUS_PATH_PORTPort class ID or GUID_All_Objects to enumerate		
DMUS_PATH_PRIMARY_BUFFER Ignored		
DMUS_PATH_SEGMENT (*) Ignored		
DMUS_PATH_SEGMENT_GRAPH (*) Ignored		
DMUS_PATH_SEGMENT_TOOL (*) Tool class ID or GUID_All_Objects to enumerate		
DMUS_PATH_SEGMENT_TRACK (*) Track class ID or GUID_All_Objects to enumerate		

Notes (*) Objects in this stage cannot be retrieved by **IDirectMusicAudioPath8::GetObjectInPath**.

(**) The standard <u>DMOs</u> provided with DirectX also support the **IMediaObject**, **IMediaObjectInPlace**, and **IMediaParams** interfaces.

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

The following example function retrieves a segment from the segment state that was created when the segment was played:

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 the **IDirectSoundBuffer8** interface in the DirectX documentation.

It is not possible to change parameters of a buffer that were set when the buffer was created by DirectMusic, such as the 3-D algorithm. However, you can specify such parameters when creating custom audiopaths in DirectMusic Producer.

Using 3-D Sound in DirectMusic

Using the DirectSound API, you can locate DirectMusic 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.

To apply 3-D effects to a buffer, you must obtain an **IDirectSound3DBuffer8** interface to a buffer that has 3-D capabilities, such as one in the <u>DMUS_APATH_DYNAMIC_3D</u> standard audiopath. You can also create a suitable audiopath from a configuration object that specifies 3-D parameters for a buffer.

The following example code creates a standard audiopath and retrieves an **IDirectSound3DBuffer8** interface. Assume that *g_pPerformance* is a valid **IDirectMusicPerformance8** pointer.

```
HRESULT
                         hr;
IDirectMusicAudioPath8*
                         g_p3DAudioPath;
IDirectSound3DBuffer8*
                         g_pDS3DBuffer;
if (SUCCEEDED(hr = g_pPerformance->CreateStandardAudioPath(
    DMUS_APATH_DYNAMIC_3D, 64, TRUE, &g_p3DAudioPath)))
{
  hr = q_p3DAudioPath->GetObjectInPath(
      DMUS_PCHANNEL_ALL, DMUS_PATH_BUFFER, 0,
      GUID_NULL, 0, IID_IDirectSound3DBuffer8,
      (LPVOID*) &g_pDS3DBuffer);
if (FAILED(hr))
{
  ErrorExit(hr); // Add error-handling code.
}
```

To adjust global 3-D parameters and manipulate the position and orientation of the listener, you must obtain an **IDirectSound3DListener8** interface from the primary buffer in any audiopath by using

IDirectMusicAudioPath8::GetObjectInPath or

IDirectMusicSegmentState8::GetObjectInPath, setting the *dwStage* parameter to DMUS_PATH_PRIMARY_BUFFER. The following example function retrieves the listener from an audiopath:

See Also

- Retrieving Objects from an Audiopath
- <u>Standard Audiopaths</u>.

Using Effects in DirectMusic

DirectX provides support for effects processing of sounds by DirectX Media Objects (DMOs). A standard set of effects is available to every DirectX application. Other DMOs can be registered on the system.

All the standard DMOs can process 8-bit or 16-bit PCM data, as well as 32-bit floating-point formats, with one or two channels at any sample rate supported by DirectSound. Waves reverberation does not support 8-bit samples.

Effects are attached to DirectSound buffers in audiopaths. To add, remove, or modify effects at run time, an application must obtain an interface to the buffer and use the DirectSound API.

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 standard WAV or MIDI files.

To apply an effect to an audiopath, you must 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 <u>Retrieving Objects from an</u> <u>Audiopath</u>. For information on how to identify standard audiopath buffers in the call to **GetObjectInPath**, see the audiopath types under <u>Standard Audiopaths</u>.

The following example code sets a standard audiopath, retrieves a buffer from the path, and sets an echo effect on the buffer:

```
// 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, &p3DAudioPath)))
  return hr;
// Get the buffer in the audiopath.
if( FAILED(hr = p3DAudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,
    DMUS_PATH_BUFFER, 0, GUID_NULL, 0, IID_IDirectSoundBuffer8,
    (LPVOID*) &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 = pDSBuffer->SetFX(1, &dsEffect, &dwResults)))
{
  p3DAudioPath->Activate(TRUE);
  return hr;
}
// You can check the value of dwResults here to ascertain
// whether the effect was allocated, and how.
// Activate the path.
p3DAudioPath->Activate(TRUE);
return hr;
```

```
© 2004 Microsoft Corporation. All rights reserved.
```

}

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 <u>Using Audiopaths</u>.

Buffer chains cannot be created by using the DirectSound API.

Using Compositional Elements

This section is a guide to using musical components in a DirectMusic application to create a soundtrack that is, to some extent, composed as it plays. It is presumed that you have a basic understanding of elements such as <u>chordmaps</u> and <u>styles</u>. If not, you should first read <u>Introduction to Dynamic Musical</u> <u>Soundtracks</u>.

It is 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
- <u>Using Styles</u>
- <u>Using Motifs</u>
- <u>Using Chordmaps</u>
- <u>Using Templates</u>
- <u>Using Transitions</u>

Music Files for Composition

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.

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 extension
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

Note Bands can be authored as part of a <u>style</u>, 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.

See Also

• Loading Audio Data

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.

Using the largest building blocks, you load highly structured segments based on styles, MIDI files, or waveforms that contain everything the performance requires to play the sound. All you have to do is load the segment and query for the <u>IDirectMusicSegment8</u> interface. Pass this interface pointer to the <u>IDirectMusicPerformance8::PlaySegment</u> or

IDirectMusicPerformance8::PlaySegmentEx method. The style playback engine selects <u>pattern</u> 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 used to build 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 <u>groove level</u> changes and embellishment patterns are to be placed, and what types of chords in the chordmap are to serve as signposts.

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 included in 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.

DirectMusic provides many options for creating transitions between segments. When you cue one segment to play after another, or to replace a currently playing segment, you have very precise control over the timing, and can synchronize the transition with the rhythm. In addition, you can have the DirectMusic composer object create a transitional measure.

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 <u>Styles</u>.

To obtain a style from a segment, use the **IDirectMusicSegment8::GetParam** method, as in the following example function, where the *dwStyleIndex* parameter specifies the index of the style within the segment:

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 possible 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.

To create a segment with a command track based on a template, call the **IDirectMusicComposer8::ComposeSegmentFromTemplate** method.

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.

See Also

• <u>Using Templates</u>

Using Motifs

A motif is a special kind of <u>pattern</u> in a <u>style</u>. A motif is intended to be played over the basic style pattern, typically in response to an interactive event. Unlike other patterns, motifs are always selected and played explicitly by the application. 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 <u>DLS</u> instrument or instruments.

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 **IDirectMusicSegment8** 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*:

```
HRESULT PlayMotif(IDirectMusicPerformance8* pPerf,
    IDirectMusicStyle8* pStyle,
    WCHAR* pwszMotifName)
{
    IDirectMusicSegment* pSeg;
    HRESULT hr;
    if ((pPerf == NULL) || (pStyle == NULL))
    {
      return E_INVALIDARG;
    }
    // Get the motif segment from the style. Check for S_OK
    // specifically, because GetMotif() returns S_FALSE if it
    // does not find the motif.
    hr = pStyle->GetMotif(pwszMotifName, &pSeg);
```

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.

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 <u>Chordmaps</u>.

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.

If a chordmap reference has been authored into a <u>style</u>, you can retrieve a pointer to its <u>IDirectMusicChordMap8</u> interface by passing its name (assigned by the author) to the <u>IDirectMusicStyle8::GetChordMap</u> method. You can also use the <u>IDirectMusicStyle8::EnumChordMap</u> method to search for a particular chordmap, or the <u>IDirectMusicStyle8::GetDefaultChordMap</u> 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**.

After 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).

See Also

• <u>Using Styles</u>

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 <u>Templates</u>.

There are two ways to obtain a template:

- Load it from a segment file or resource, and request the **<u>IDirectMusicSegment8</u>** interface.
- Create it from a shape, using the <u>IDirectMusicComposer8::ComposeTemplateFromShape</u> 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 from which you can obtain IDirectMusicSegment8.

After you have obtained a template segment object, you can pass it to the **IDirectMusicComposer8::ComposeSegmentFromTemplate** method, along with pointers to a <u>style</u> 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 **IDirectMusicSegment** interface.

See Also

• Loading Audio Data

Using Transitions

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.

You have your choice of three techniques for composing transitional segments:

- 89The **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 <u>chordmap</u>, a command, and a set of flags as parameters:

- The chordmap 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 <u>pattern</u>—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 <u>style</u>.
- The flags are from the **DMUS_COMPOSEF_FLAGS** enumeration 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 cannot 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.

DirectMusic also provides many options for controlling the timing of transitions from one segment to another. For more information, see the following topics:

- <u>Segment Timing</u>
- <u>DMUS_SEGF_FLAGS</u>

Track Composition

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 for composition.

One standard track, the signpost track, supports composition. The signpost track composes a new chord track from a <u>chordmap</u>.

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**.

Effect
The track is composed by the
IDirectMusicSegment8::Compo
method.
The track is automatically
E composed each time the segment
loops.
The track is automatically
E 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**.

Using Audio Scripts

Scripts are collections of variables and routines authored in DirectMusic Producer. Although they consist mainly of text, scripts also contain a few binary parameters. For a conceptual overview, see <u>Audio Scripts</u>.

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 example function loads and initializes a script.

```
HRESULT LoadScript(IDirectMusicPerformance8 * pPerf,
                                 IDirectMusicLoader8* pLoader,
                                 WCHAR* wstrFileName,
                                 IDirectMusicScript8** ppScript)
{
  DMUS SCRIPT ERRORINFO errInfo;
  HRESULT hr;
  if ((NULL == pPerf) || (NULL == pLoader))
  {
    return E_INVALIDARG;
  }
  if (SUCCEEDED(hr = pLoader->LoadObjectFromFile(
    CLSID_DirectMusicScript, IID_IDirectMusicScript8,
    wstrFileName, (LPVOID*) ppScript)))
  {
    if (FAILED(hr = (*ppScript)->Init(pPerf, &errInfo)))
    {
      (*ppScript)->Release();
    }
  }
  return hr;
}
```

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.

Capturing MIDI

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.
```

```
// pb[0] is the status byte.
    // pb[1] and pb[2] are the data bytes.
    }
    else if (hr == S_FALSE)
    {
        break; // No more messages in the buffer.
    }
} // Done with the buffer.
} // Done reading pending events.
```

If you don't want 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.

Advanced Topics in DirectMusic

This section contains information needed for specialized applications that need functionality beyond that covered under <u>Using DirectMusic</u>.

Information is presented in the following topics:

- <u>DirectMusic Tracks</u>
- <u>Using DirectMusic Messages</u>
- <u>Using DirectMusic Ports</u>
- <u>Custom Loading</u>
- <u>Using Instrument Collections</u>
- Low-Level DLS
- <u>DirectMusic Tools</u>
- <u>Property Sets for DirectMusic Ports</u>

DirectMusic Tracks

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 <u>Message Creation and</u> <u>Delivery</u> 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
- <u>Track Configuration</u>

Standard Track Types

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 messages of type DMUS_I DMUS_TRANSPOSE_PI DMUS_CHANNEL_PRI DMUS_MIDI_PMSG (for Used in segments based on
Chord	CLSID_DirectMusicChordTrack	Used to convert <u>music valu</u> values. Sends messages of <u>DMUS_NOTIFICATION</u> GUID_NOTIFICATION_C
Chordma	pCLSID_DirectMusicChordMapTrack	Used in <u>template</u> segments tracks.
Comman	d CLSID_DirectMusicCommandTrack	Used in template segments tracks, and in style segmen patterns are played. Sends I DMUS_NOTIFICATION GUID_NOTIFICATION_C notifications.
Lyrics	CLSID_DirectMusicLyricsTrack	Used to synchronize words Generates messages of type DMUS_LYRIC_PMSG.
Marker	CLSID_DirectMusicMarkerTrack	Used for flow control. The valid start times for the seg at which new segments can information, see <u>Segment 1</u>
		Used to play motifs to acco segments. Sends messages

Motif	CLSID_DirectMusicMotifTrack	DMUS_CURVE_PMSG, DMUS_NOTE_PMSG, ar DMUS_NOTIFICATION GUID_NOTIFICATION_M notifications).
Mute	CLSID_DirectMusicMuteTrack	Enables <u>performance chann</u> muted. Used with either sty based segments.
Parameter control	CLSID_DirectMusicParamControlTrack	Controls the settings on toc other objects that support tl interface.
Pattern	CLSID_DirectMusicPatternTrack	Contains a single musical p track is similar to a sequent music values rather than fix makes it possible to auditio different chords, and is use authoring applications. It m play an accompaniment. It own interface, IDirectMus
Script	CLSID_DirectMusicScriptTrack	Calls routines in an audio s
Segment trigger	CLSID_DirectMusicSegmentTriggerTrack	Triggers the playback of se enables the author of a file within a segment, rather tha the application developer.
Sequence	CLSID_DirectMusicSeqTrack	Sends sequence messages c DMUS_NOTE_PMSG an DMUS_MIDI_PMSG. Us on MIDI files. Also sends r DMUS_CURVE_PMSG f the .sgt format.
Signpost	CLSID_DirectMusicSignPostTrack	Used in template segments tracks.
Style	CLSID_DirectMusicStyleTrack	Fundamental track for segn Sends messages of type DMUS_TIMESIG_PMS(DMUS_CURVE_PMSG, DMUS_NOTE_PMSG, ar DMUS_NOTIFICATION

		GUID_NOTIFICATION_N notifications).
SysEx	CLSID_DirectMusicSysExTrack	Sends system exclusive me <u>DMUS_SYSEX_PMSG</u> . U based on MIDI files.
Tempo	CLSID_DirectMusicTempoTrack	Controls the tempo of the p
Time Signature	CLSID_DirectMusicTimeSigTrack	Sends messages of type DMUS_TIMESIG_PMSC GUID_NOTIFICATION_N notifications. The time sign imported MIDI files and au specifically created with on style track implements the 1 functionality, so it is not ne that contains a style track to signature track as well. The Time Signature track s IDirectMusicTrack interfa IDirectMusicTrack8 .
Wave	CLSID_DirectMusicWaveTrack	Sends messages of type DN to play time-stamped wave

Track Configuration

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 enabled again 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 effect 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 only for that segment state. To ensure that the behavior 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 the track is disabled 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);

See Also

- <u>Track Composition</u>
- <u>Control Segments</u>
- <u>Self-Controlling Segments</u>

Using DirectMusic Messages

Data passes through the DirectMusic performance and to the synthesizer 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.

Note The **<u>DMUS_MIDI_PMSG</u>** structure contains data equivalent to that in any standard MIDI message. However, it is used in performance messages.

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 <u>DirectMusic Tools</u>. 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.

The following topics discuss messages and how they are routed:

- <u>Channels</u>
- Message Creation and Delivery
- <u>Application-Created Messages</u>
- <u>Performance Message Types</u>
- <u>Curves</u>
- <u>MIDI Messages</u>

See Also

- <u>Overview of Audio Data Flow</u>
- Notification and Event Handling

Channels

A channel is a destination for messages that are specific to one or more parts. A part might contain notes for a single instrument, or it might contain one or more waveforms.

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 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 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 plays on a 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 waveform 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 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. Applications and synthesizers can set their own channel priorities.

See Also

- <u>DMUS_CHANNEL_PRIORITY_PMSG</u>
- IDirectMusicPort8::GetChannelPriority

Message Creation and Delivery

When a segment is played, most of its tracks generate messages containing data about events that are scheduled to take place during playback. For more information, see <u>DirectMusic Tracks</u>.

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 <u>Disabling and Enabling Track Parameters</u>.

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 <u>performance channel</u> 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.

Note 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.

Messages are first sent to any tools in the segment toolgraph, then to the audiopath toolgraph, and finally to the performance toolgraph. 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. 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.

Finally, the message 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.

See Also

- <u>Overview of Audio Data Flow</u>
- <u>DirectMusic Tools</u>

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 <u>DLS</u> instrument.

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 <u>performance channel</u> 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**.

See Also

• <u>IDirectMusicPerformance8::SendPMsg</u>

Performance Message Types

Messages are stored in various structures derived from **DMUS_PMSG**. Because the C programming language 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.

Structure	Content
DMUS_PMSG	Simple message with no additional
	parameters.
DMUS_CHANNEL_PRIORITY_PMS(Channel priority change. See
DWO5_CHAINEL_PRIORITI_PMSC	<u>Channels</u> .
DMUS_CURVE_PMSG	<u>Curve</u> .
DMUS_LYRIC_PMSG	Text.
	Any MIDI message that does not
DMUS_MIDI_PMSG	have a unique message type—for
	example, a control change.
	Music note. MIDI note-on and note-
DMUS NOTE PMSG	off messages are combined in this
DMOS_NOTL_TMSG	structure, which specifies the
	duration of the note.
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	Waveform playback.

The following standard message structures are defined.

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.

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 the shape of the curve. 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 example function 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.

```
HRESULT SendCurveMsg(IDirectMusicPerformance8* pPerf)
{
    DMUS_CURVE_PMSG *pCurveMsg;
    HRESULT hr;
    if (NULL == pPerf) return E_INVALIDARG;
    hr = pPerf->AllocPMsg(sizeof(DMUS_CURVE_PMSG),
        (DMUS_PMSG**) &pCurveMsg);
    if (SUCCEEDED(hr))
```

```
{
    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 = 0xFFFFFF;
    pCurveMsg->mtDuration = 5000;
    pCurveMsg->nEndValue = 0;
    pCurveMsg->bCurveShape = DMUS_CURVES_LINEAR;
    pCurveMsq->bCCData = 7;
    pCurveMsg->bFlags = DMUS_CURVE_RESET | DMUS_CURVE_START_FROM_CUR
    pCurveMsg->bType = DMUS_CURVET_CCCURVE ;
    pCurveMsg->mtResetDuration = 0;
    pCurveMsg->nResetValue = 127;
    hr = pPerf->SendPMsq((DMUS_PMSG*) pCurveMsq);
  }
  return hr;
}
```

Note A simpler way to implement volume fading is by using **IDirectMusicAudioPath8::SetVolume**. This method always uses the linear shape for the curve.

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
- <u>MIDI System Messages</u>

MIDI Channel Messages

A channel message is addressed to a particular MIDI channel, which corresponds to a part in the music.

A channel message can be either a *mode message* or a *voice message*.

A mode message specifies how the channel is to respond to subsequent voice messages. For example, a mode message might instruct the channel to remain silent, ignoring all note-on messages.

A voice message instructs the channel to begin or stop playing a note or to modify notes in some way. Most channel messages are voice messages.

Voice message	Command
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.

The following table describes types of voice messages.

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 that specifies the duration of the note. Performance messages also contain additional information about timing and routing. The following topics contain more information about MIDI channel messages and how they are implemented in DirectMusic:

- <u>MIDI Notes</u>
- Program Changes
- Bank Selection
- <u>DirectMusic Patch Numbers</u>

MIDI Notes

The data bytes of a note-on message represent the pitch and <u>velocity</u>. 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.

For information on how DirectMusic converts to and from MIDI notes, see <u>Music Values and MIDI Notes</u>.

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 seven 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 two 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

In DirectMusic, the instrument patch number is not the seven-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.

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

The organization of DirectMusic patch values is shown in the following table.

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
System real-time	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.

In DirectMusic, the **DMUS_SYSEX_PMSG** structure contains the length of the data and a pointer to an array of data bytes.

Using DirectMusic Ports

This section covers access to DirectMusic ports, which is usually required only by applications that do not use audiopaths. 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 <u>DLS</u> instruments, read incoming data, and cue playback buffers.

If your application initializes the performance by using **IDirectMusicPerformance8::InitAudio**, as is recommended, the audiopath manages ports and the mapping of <u>performance channels</u> to ports. You can obtain an interface to a port in the audiopath by using the **IDirectMusicPerformance8::PChannelInfo** method.

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 example function demonstrates how an object might be created for the default port, setting five channel groups on the port.

```
HRESULT CreateTypicalPort(IDirectMusic8* pDM)
{
    IDirectMusicPort8* pPort;
    DMUS_PORTPARAMS dmos;
    if (NULL == pDM) return E_INVALIDARG;
    ZeroMemory(&dmos, sizeof(DMUS_PORTPARAMS));
    dmos.dwSize = sizeof(DMUS_PORTPARAMS);
    dmos.dwValidParams = DMUS_PORTPARAMS_CHANNELGROUPS;
    dmos.dwChannelGroups = 5;
    return pDM->CreatePort(GUID_NULL, &dmos, &pPort, NULL);
}
```

}

After creating a port, you must activate it by calling **IDirectMusic8::Activate** or **IDirectMusicPort8::Activate** and then attach it to the performance by using the **IDirectMusicPerformance8::AddPort** method.

When you add a port to a performance, assign a block of <u>performance channels</u> to it by calling the <u>IDirectMusicPerformance8::AssignPChannelBlock</u> 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:

- <u>Default Port</u>
- <u>Property Sets for DirectMusic Ports</u>

Default Port

Under versions of Windows 98 prior to Windows 98 Second Edition, and always when hardware that supports <u>DLS</u> is not available, the Microsoft software synthesizer is the default port. Under other versions of Windows, 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

It might happen that an application needs to manage object loading itself—for example, because all objects are stored in a special compressed content file. The application can create its own loader in the form of an object that supports the **IDirectMusicLoader8** interface, with at least the **GetObject** method implemented. This implementation of the loader must also create its own stream object that has both the **IStream** and the **IDirectMusicGetLoader8** interfaces.

When **GetObject** receives a request to load an object from a file or resource, it creates a stream and passes the **IStream** pointer to the object's **IPersistStream::Load** method. When it receives a request to load an object from an existing stream created by the application to manage reading from a custom file, it creates a copy of the **IStream** with the same seek pointer and passes this copy to **Load**.

The stream object's implementation of **IDirectMusicGetLoader8::GetLoader** is used by loadable objects to retrieve a pointer to the loader that created the **IStream**. Objects need this pointer in order to call **GetObject** recursively when they find references to other objects. For example, a segment object might contain references to WAV files, which must be loaded along with the segment.

To support container objects, the loader must also implement the IDirectMusicLoader8::SetObject method. The implementation retains all information in the supplied DMUS_OBJECTDESC structure, copying the stream pointer if necessary. It then creates the object and calls IDirectMusicObject8::ParseDescriptor to obtain the rest of the object's descriptive information. However, the loader should not actually load the object until GetObject is called. If only containers with embedded objects need to be handled, only the case where DMUS_OBJ_STREAM is set needs to be implemented.

For more information, see <u>Custom Loading in DirectMusic</u> at msdn.microsoft.com.

^{© 2004} Microsoft Corporation. All rights reserved.

Using Instrument Collections

In most applications, <u>DLS</u> instrument data is associated with bands and is downloaded to the synthesizer when the band is downloaded. For more information, see <u>Using Bands</u>.

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
- <u>Working with Instruments</u>

Loading and Downloading Collections

Collections are loaded like other objects. To load the standard Roland 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.

The following example function illustrates how to load a collection identified by its GUID:

When 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 <u>Playing a MIDI File with Custom Instruments</u>.

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 <u>DLS</u> data needed by the instruments in that band is downloaded.

See Also

- Loading Audio Data
- <u>Using Bands</u>

Working with Instruments

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 example function enumerates all instruments in a collection:

```
void ListInstruments(IDirectMusicCollection8 *pCollection)
{
  HRESULT hr = S OK;
  DWORD dwPatch;
  WCHAR wszName[MAX_PATH];
  DWORD dwIndex;
  for (dwIndex = 0; hr == S_0K; dwIndex++)
  {
    hr = pCollection->EnumInstrument(
      dwIndex, &dwPatch, wszName, MAX_PATH);
    if (hr == S_0K)
    {
      // Do something with name of patch in 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 **<u>IDirectMusicInstrument8::SetPatch</u>** 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 waveforms and <u>articulation</u> 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 DownloadCollection(
    IDirectMusicCollection8 *pCollection,
    IDirectMusicPort8 *pPort,
    IDirectMusicDownloadedInstrument8 **ppDLInstrument,
    DWORD dwPatch,
    DWORD dwLowNote,
    DWORD dwHighNote)
 {
  HRESULT hr;
  if ((NULL == pCollection) || (NULL == pPort) || (NULL == ppDLInstr
  {
    return E INVALIDARG;
  }
  IDirectMusicInstrument8* pInstrument;
  hr = pCollection->GetInstrument(dwPatch, &pInstrument);
  if (SUCCEEDED(hr))
  {
    DMUS_NOTERANGE NoteRange[1];
    NoteRange[0].dwLowNote = dwLowNote;
    NoteRange[0].dwHighNote = dwHighNote;
    hr = pPort->DownloadInstrument(pInstrument, ppDLInstrument, Note
    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 DownloadAndUnload(
    IDirectMusicInstrument8* pInstrument,
    IDirectMusicPort8 *pPort)
{
    HRESULT hr;
    IDirectMusicDownloadedInstrument* pDLInstrument;
    if ((NULL == pInstrument) || (NULL == pPort)) return E_INVALIDARG;
    hr = pPort->DownloadInstrument(pInstrument, &pDLInstrument, NULL,
    if (SUCCEEDED(hr))
    {
        hr = pPort->UnloadInstrument(pDLInstrument);
        pDLInstrument->Release();
    }
    return hr;
}
```

Low-Level DLS

If you are writing an application that edits <u>DLS</u> collections or creates instruments from waveform samples at run time, you must 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, 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;
```

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, you will create the following buffers:

- One instrument buffer, which represents the entire instrument definition with all the regions and <u>articulations</u>.
- One or more waveform buffers, which describe each waveform 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 waveforms. 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 waveforms, 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 waveform.
- 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 WAV 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:

- <u>DMUS_ARTICPARAMS</u>
- <u>DMUS_ARTICULATION</u>
- <u>DMUS_ARTICULATION2</u>
- <u>DMUS_COPYRIGHT</u>
- <u>DMUS_EXTENSIONCHUNK</u>
- <u>DMUS_INSTRUMENT</u>
- <u>DMUS_NOTERANGE</u>
- DMUS_REGION

The WAV 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 waveform 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 waveform 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 waveforms. 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 waveform buffer, take one extra step. Create both a new waveform buffer and an updated instrument buffer that references it. Download the new waveform, then the new instrument. Then unload the old instrument and the old waveform.

More information is contained in the following topic:

• <u>Creating DLS Instruments Programmatically</u>

Creating DLS Instruments Programmatically

The following example code shows how to create a DLS instrument from a WAV file and make it available to the performance.

The entry point for the example is the PlayDLSFromWAV function, which creates and downloads the instrument and plays two notes on it by sending performance messages. This function uses two classes defined in the DirectX sample framework. The *pWaveFile* parameter is an instance of **CWaveFile** representing a WAV file that has already been opened. An instance of **CMusicManager** is used to create and initialize the performance.

```
// Declare data structures for download.
#pragma pack(2)
struct INSTRUMENT_DOWNLOAD
{
    DMUS DOWNLOADINFO
                        dlInfo;
                        ul0ffsetTable[4];
    ULONG
    DMUS_INSTRUMENT
                        dmInstrument;
                        dmRegion;
    DMUS REGION
                        dmArticulation;
    DMUS ARTICULATION
    DMUS_ARTICPARAMS
                        dmArticParams;
};
struct WAVE_DOWNLOAD
{
    DMUS DOWNLOADINFO
                        dlInfo;
    ULONG
                        ulOffsetTable[2];
    DMUS WAVE
                        dmWave;
    DMUS WAVEDATA
                        dmWaveData;
};
#pragma pack()
// Define some values for instrument parameters.
#define FIVE_HERTZ (0xFCACAE9C)
#define ZERO SECONDS (0x80000000)
#define ONE_MILLISECOND (0xD1490F12)
#define ONE_HUNDRED_PERCENT (0x03E80000)
```

```
void InitializeInstDownload(INSTRUMENT_DOWNLOAD *pInstDownload, DWOR
{
    ZeroMemory(pInstDownload, sizeof(INSTRUMENT_DOWNLOAD));
    pInstDownload->dlInfo.dwDLType = DMUS_DOWNLOADINFO_INSTRUMENT;
    pInstDownload->dlInfo.cbSize = sizeof(INSTRUMENT_DOWNLOAD);
    pInstDownload->dlInfo.dwDLId = dwDLId;
    pInstDownload->dlInfo.dwNumOffsetTableEntries = 4;
    pInstDownload->ulOffsetTable[0] = offsetof(INSTRUMENT_DOWNLOAD,d
    pInstDownload->ulOffsetTable[1] = offsetof(INSTRUMENT_DOWNLOAD,d
    pInstDownload->ulOffsetTable[2] = offsetof(INSTRUMENT_DOWNLOAD,d
    pInstDownload->ulOffsetTable[3] = offsetof(INSTRUMENT_DOWNLOAD,d
    pInstDownload->dmInstrument.ulFirstRegionIdx = 1;
    pInstDownload->dmInstrument.ulGlobalArtIdx = 2;
    pInstDownload->dmInstrument.ulPatch = dwPatch;
    pInstDownload->dmRegion.RangeKey.usHigh = 127;
    pInstDownload->dmRegion.RangeVelocity.usHigh = 127;
    pInstDownload->dmRegion.fusOptions = F RGN OPTION SELFNONEXCLUSI
    pInstDownload->dmRegion.WaveLink.ulChannel = 1;
    pInstDownload->dmRegion.WaveLink.ulTableIndex = dwDLIdWave;
    pInstDownload->dmRegion.WaveLink.usPhaseGroup = 0;
    pInstDownload->dmRegion.WSMP.cbSize = sizeof(WSMPL);
    pInstDownload->dmRegion.WSMP.fulOptions = F_WSMP_NO_TRUNCATION;
    pInstDownload->dmRegion.WSMP.usUnityNote = 60; // Middle C
    pInstDownload->dmRegion.WLOOP[0].cbSize = sizeof(WLOOP);
    pInstDownload->dmRegion.WL00P[0].ulType = WL00P_TYPE_FORWARD;
    pInstDownload->dmArticulation.ulArt1Idx = 3;
    pInstDownload->dmArticParams.LF0.tcDelay = ZER0 SECONDS;
    pInstDownload->dmArticParams.LF0.pcFrequency = FIVE_HERTZ;
    pInstDownload->dmArticParams.PitchEG.tcAttack = ZERO_SECONDS;
    pInstDownload->dmArticParams.PitchEG.tcDecay = ZERO SECONDS;
    pInstDownload->dmArticParams.PitchEG.ptSustain = ONE_HUNDRED_PER
    pInstDownload->dmArticParams.PitchEG.tcRelease = ONE_MILLISECOND
    pInstDownload->dmArticParams.VolEG.tcAttack = ZERO_SECONDS;
    pInstDownload->dmArticParams.VolEG.tcDecay = ZER0_SECONDS;
    pInstDownload->dmArticParams.VolEG.ptSustain = ONE HUNDRED PERCE
    pInstDownload->dmArticParams.VolEG.tcRelease = ONE MILLISECOND;
}
void InitializeWaveDownload(WAVE_DOWNLOAD *pWaveDownload, DWORD dwDL
{
    ZeroMemory(pWaveDownload, sizeof(WAVE_DOWNLOAD));
    pWaveDownload->dlInfo.dwDLType = DMUS_DOWNLOADINFO_WAVE;
    pWaveDownload->dlInfo.cbSize = dwOverallSize;
```

```
pWaveDownload->dlInfo.dwDLId = dwDLId;
```

```
pWaveDownload->dlInfo.dwNumOffsetTableEntries = 2;
    pWaveDownload->ulOffsetTable[0] = offsetof(WAVE_DOWNLOAD, dmWave)
    pWaveDownload->ulOffsetTable[1] = offsetof(WAVE_DOWNLOAD,dmWaveD
    pWaveDownload->dmWave.ulWaveDataIdx = 1;
    memcpy(&pWaveDownload->dmWave.WaveformatEx, pwfex, sizeof(WAVEF0
    pWaveDownload->dmWaveData.cbSize = dwWaveSize;
}
void PlayDLSFromWAV(HWND hWndMain, CWaveFile *pWaveFile)
{
    const DWORD dwPatch = 0x00123456;
    // Create and initialize performance.
    CMusicManager musicManager;
    IDirectMusicPerformance8* pPerf;
    HRESULT hr = musicManager.Initialize(hWndMain);
    pPerf = musicManager.GetPerformance();
    // Get interfaces to the port.
    IDirectMusicPort* pIDirectMusicPort = NULL;
    if (SUCCEEDED(hr))
    {
        hr = pPerf->PChannelInfo(0, &pIDirectMusicPort, NULL, NULL);
    }
    IDirectMusicPortDownload8* pIDirectMusicPortDownload8 = NULL;
    if (SUCCEEDED(hr))
    ł
        hr = pIDirectMusicPort->QueryInterface(IID IDirectMusicPortD
    }
    // Reserve two download IDs, and retrieve the first.
    DWORD dwDLId = 0;
    if (SUCCEEDED(hr))
    {
        hr = pIDirectMusicPortDownload8->GetDLId(&dwDLId, 2);
    }
    // Allocate a buffer for the instrument data (regions, articulat
    IDirectMusicDownload8* pIDirectMusicDownloadArticulation = NULL;
    if (SUCCEEDED(hr))
```

```
{
    hr = pIDirectMusicPortDownload8->AllocateBuffer( sizeof(INST
}
// Allocate a buffer for the WAV data.
IDirectMusicDownload8* pIDirectMusicDownloadWave = NULL;
DWORD dwAppend = 0;
if (SUCCEEDED(hr))
{
    hr = pIDirectMusicPortDownload8->GetAppend(&dwAppend);
    if (SUCCEEDED(hr))
    {
        hr = pIDirectMusicPortDownload8->AllocateBuffer(sizeof(\u00eb
                dwAppend * pWaveFile->GetFormat()->nBlockAlign +
                &pIDirectMusicDownloadWave);
    }
}
// Read format data from the WAV file into the buffer, and downl
void *pvData = NULL;
DWORD dwSize = 0;
if (SUCCEEDED(hr))
{
    hr = pIDirectMusicDownloadWave->GetBuffer( &pvData, &dwSize
    if (SUCCEEDED(hr))
    {
        InitializeWaveDownload((WAVE_DOWNLOAD*)pvData, dwDLId, p
                pWaveFile->GetSize(), dwSize);
        DWORD dwRead = 0;
        hr = pWaveFile->Read(((WAVE_DOWNLOAD*)pvData)->dmWaveDat
                pWaveFile->GetSize(), &dwRead);
        if (SUCCEEDED(hr) && pWaveFile->GetSize() == dwRead)
        {
            hr = pIDirectMusicPortDownload8->Download(pIDirectMu
            if (hr == DMUS E NOTMONO)
            {
              MessageBox(hWndMain, "WAV must be mono.", "Error",
            }
        }
    }
}
// Put instrument data into the buffer and download to the port.
if (SUCCEEDED(hr))
{
    hr = pIDirectMusicDownloadArticulation->GetBuffer( &pvData,
```

```
if (SUCCEEDED(hr))
    {
        InitializeInstDownload((INSTRUMENT_DOWNLOAD *)pvData, dw
        hr = pIDirectMusicPortDownload8->Download(pIDirectMusicD
    }
}
// Get the performance toolgraph so messages can be stamped.
IDirectMusicGraph* pIDirectMusicGraph = NULL;
if (SUCCEEDED(hr))
{
    hr = pPerf->QueryInterface(IID_IDirectMusicGraph, (void **)&
}
// Create and send a message to put the instrument on channel 0.
DMUS PATCH PMSG *pDMUS PATCH PMSG = NULL;
if (SUCCEEDED(hr))
{
    hr = pPerf->AllocPMsq( sizeof(DMUS PATCH PMSG), (DMUS PMSG *
}
if (SUCCEEDED(hr))
{
    pDMUS_PATCH_PMSG->dwType = DMUS_PMSGT_PATCH;
    pDMUS PATCH PMSG->dwPChannel = 0;
    pDMUS_PATCH_PMSG->dwFlags = DMUS_PMSGF_REFTIME ;
    pDMUS_PATCH_PMSG->byInstrument = dwPatch & 0x7F;
    pDMUS PATCH PMSG->byLSB = (dwPatch & 0x7f00) >> 8;
    pDMUS_PATCH_PMSG->byMSB = (dwPatch & 0x7f0000) >> 16;
    hr = pIDirectMusicGraph->StampPMsq((DMUS PMSG *)pDMUS PATCH
}
if (SUCCEEDED(hr))
{
    hr = pPerf->SendPMsq((DMUS_PMSG *)pDMUS_PATCH_PMSG);
    if (FAILED(hr))
    {
        pPerf->FreePMsq((DMUS_PMSG *)pDMUS_PATCH_PMSG);
    }
}
// The instrument is now available to be played. The following c
// plays two "notes" at different pitches and durations.
DMUS NOTE PMSG * pDMUS NOTE PMSG = NULL;
if (SUCCEEDED(hr))
{
    hr = pPerf->AllocPMsg( sizeof(DMUS_NOTE_PMSG), (DMUS_PMSG **
}
```

```
if (SUCCEEDED(hr))
{
    pDMUS_NOTE_PMSG->dwType = DMUS_PMSGT_NOTE;
    pDMUS NOTE PMSG->dwPChannel = 0;
    pDMUS_NOTE_PMSG->dwFlags = DMUS_PMSGF_REFTIME;
    pDMUS_NOTE_PMSG->bFlags = DMUS_NOTEF_NOTEON;
    pDMUS_NOTE_PMSG->bVelocity = 127;
    pDMUS_NOTE_PMSG->bMidiValue = 60;
    pDMUS_NOTE_PMSG->mtDuration = DMUS_PPQ * 4; // Whole note
    hr = pIDirectMusicGraph->StampPMsq((DMUS PMSG *)pDMUS NOTE F
}
if (SUCCEEDED(hr))
{
    hr = pPerf->SendPMsg((DMUS_PMSG *)pDMUS_NOTE_PMSG);
    if (FAILED(hr))
    {
        pPerf->FreePMsg((DMUS_PMSG *)pDMUS_NOTE_PMSG);
    }
}
Sleep(1000);
DMUS_NOTE_PMSG *pDMUS_NOTE_PMSG2 = NULL;
if (SUCCEEDED(hr))
{
    hr = pPerf->AllocPMsg( sizeof(DMUS_NOTE_PMSG), (DMUS_PMSG **
if (SUCCEEDED(hr))
{
    pDMUS NOTE PMSG2->dwType = DMUS PMSGT NOTE;
    pDMUS_NOTE_PMSG2->dwPChannel = 0;
    pDMUS_NOTE_PMSG2->dwFlags = DMUS_PMSGF_REFTIME;
    pDMUS_NOTE_PMSG2->bFlags = DMUS_NOTEF_NOTEON;
    pDMUS_NOTE_PMSG2->bMidiValue = 70;
    pDMUS_NOTE_PMSG2->bVelocity = 127;
    pDMUS_NOTE_PMSG2->mtDuration = DMUS_PPQ * 2; // Half note
    hr = pIDirectMusicGraph->StampPMsg((DMUS_PMSG *)pDMUS_NOTE_P
}
if (SUCCEEDED(hr))
Ł
    hr = pPerf->SendPMsg((DMUS_PMSG *)pDMUS_NOTE_PMSG2);
    if (FAILED(hr))
    {
        pPerf->FreePMsg((DMUS_PMSG *)pDMUS_NOTE_PMSG2);
    }
}
// Allow time for second note to play before music manager goes
```

```
// and shuts down performance.
```

```
Sleep(4000);
// Clean up. SAFE_RELEASE is defined in dmutil.h.
SAFE_RELEASE(pIDirectMusicPort);
SAFE_RELEASE(pIDirectMusicPortDownload8);
SAFE_RELEASE(pIDirectMusicDownloadArticulation);
SAFE_RELEASE(pIDirectMusicDownloadWave);
SAFE_RELEASE(pIDirectMusicGraph);
```

© 2004 Microsoft Corporation. All rights reserved.

}

DirectMusic Tools

A DirectMusic 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. This tool converts performance messages to standard MIDI messages and streams them to the synthesizer. Other tools are implemented by the application or obtained from libraries.

The following topics provide more information on tools:

- Creating a Tool
- Implementing a Tool in the Client Application

See Also

• **Overview of Audio Data Flow**

Creating a Tool

A tool object can reside in a DLL or in the client application. The object implements the methods of **IDirectMusicTool** or **IDirectMusicTool8**. The purpose of each method is summarized in the following table.

Method	Description	
Init	Performs any needed initialization of the object. This method is called when the application adds the tool to a graph by calling IDirectMusicGraph8::InsertTool . It should always return S_OK.	
<u>GetMsgDeliveryType</u>	Specifies when the performance should deliver messages to the tool by calling its ProcessPMsg method. Most tools can specify DMUS_PMSGF_TOOL_IMMEDIATE.	
<u>GetMediaTypes</u>	Returns an array of message types that the tool processes. This method is called by the DirectMusic performance to determine which messages to pass to the ProcessPMsg method.	
<u>GetMediaTypeArraySize</u> Specifies the number of message types in the array.		
<u>ProcessPMsg</u>	Processes each message. This method is called by the performance each time a message that matches the requested types is available. After processing, the method either requeues or deletes the message.	
<u>Flush</u>	Specifies the behavior of the tool when it receives a message sent as a result of an invalidation. This can happen, for example, when a note or curve is in progress and the segment stops unexpectedly. Most tools can simply requeue the message.	

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. When 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.

The following sample code from the tool's implementation of **IDirectMusicTool8::ProcessPMsg** stamps the message (*pPMsg*) for the next tool, or frees the message if this is not possible.

It's important to stamp the message before reaching any code paths that might cause the method to return. If the message is not stamped for the next tool when it is handed back to the queue, it will be sent back to this tool, possibly resulting in an endless loop. On the other hand, be aware that **StampPMsg** can change the value in the **dwPChannel** member of the message structure. If your tool uses this value, save it before stamping the message.

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.

See Also

- <u>Music Tool Sample</u>
- <u>DirectMusic Tool Wizard</u>

Implementing a Tool in the Client Application

All tools other than the output tool are collected in toolgraphs. Even if your application is using only a single 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.

Note Do not use or distribute tools from non-trusted sources. Tools can contain unsafe code.

The following sample code is from a client that uses a tool in a DLL. First, the client creates an object from a known class identifier and a known interface identifier. It then obtains the **IDirectMusicTool8** interface, creates a graph, and inserts the tool in the graph. It is assumed that the **ILyricsReader** interface, together with the CLSID and IID, is declared in an included tool header.

```
HRESULT SetupLyricsTool(IDirectMusicPerformance8* pPerf)
{
  ILyricsReader* pLyricsReader;
  IDirectMusicTool* pTool;
  IDirectMusicGraph* pGraph;
  HRESULT hr;
  if (SUCCEEDED(hr = CoCreateInstance(CLSID_LyricsReader, NULL,
                CLSCTX_INPROC_SERVER, IID_ILyricsReader,
               (void **) &pLyricsReader)))
  {
    if (SUCCEEDED(hr = pLyricsReader->QueryInterface(IID_IDirectMusi
      (void**)&pTool)))
    {
      if ( SUCCEEDED(hr = CoCreateInstance( CLSID_DirectMusicGraph,
          CLSCTX_INPROC, IID_IDirectMusicGraph, (void**)&pGraph)))
      {
        if (SUCCEEDED(pGraph->InsertTool(pTool, NULL, 0, 0 )))
        {
          hr = pPerf->SetGraph(pGraph);
        pGraph->Release();
      }
    }
```

```
}
return hr;
}
```

The tool will now process messages from all segments in the performance. To restrict the application of the tool to a particular segment, use **IDirectMusicSegment8::SetGraph** instead.

Property Sets for DirectMusic Ports

Through property sets, DirectMusic is able to support extended services offered by synthesizers.

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. Individual properties may also have associated parameters. The meaning of the parameters is defined along with the properties.

Use the **IKsControl::KsProperty** method to find out whether a property is available and then to set and retrieve values for that property. You can 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:

```
#include <dmksctrl.h>
```

```
BOOL IsGMSupported(IDirectMusicPort8 *pPort)
```

```
{
  HRESULT
          hr;
              *pControl;
  IKsControl
  KSPROPERTY ksp;
  DWORD
         dwFlags;
  ULONG
          cb;
          fIsSupported;
  BOOL
  hr = pPort->QueryInterface(IID_IKsControl, (void**)&pControl);
  if (FAILED(hr))
  {
    // Port does not support properties; assume no GM support.
    return FALSE;
  }
  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;
}
```

Related Software

This section contains brief descriptions of software distributed with the DirectX SDK that can be used in the development of DirectMusic applications. The following software is described:

- <u>DirectMusic Producer</u>
- DirectMusic Style Library
- <u>DirectMusic Tool Wizard</u>

DirectMusic Producer

Microsoft DirectMusic Producer is a powerful authoring application included with the DirectX SDK. Its enables composers and sound designers to create most elements of a dynamic soundtrack, including the following:

- Audiopath configurations
- Bands
- Chordmaps
- Downloadable Sounds collections
- Scripts
- Toolgraphs
- Styles and style-based segments
- MIDI-based segments
- Waveform-based segments

For more information, see the DirectMusic Producer Help.

DirectMusic Style Library

The DirectX SDK includes many sample DirectMusic styles and an application, StylePlayer, that enables you to audition these styles with different chordmaps, bands, shapes, and motifs.

The style library and StylePlayer are found in the Essentials folder on the DirectX CD. To use them, copy the DirectMusic Style Library folder to your hard disk.

For more information, see the StylePlayer.txt file.

DirectMusic Tool Wizard

The DirectMusic Tool Wizard simplifies the task of creating a custom tool that can be inserted in DirectMusic Producer or in a DirectMusic application to intercept and process performance messages.

Microsoft Visual C++® 6.0 is required to run the wizard. Select **New** from the **File** menu, and then select DirectMusic Tool Wizard from the **Projects** tab of the **New** dialog box.

The wizard creates a ready-to-compile project containing C++ code that handles the basic COM component creation and DLL registration as well as the entry point for the DLL. In addition, the wizard generates the tool's implementation class with basic services already in place. After the project is created, you must add code that implements the tool's functionality.

For more information, see the Help for the wizard.

DirectMusic C++ Samples

The sample applications described in this section demonstrate the use and capabilities of the Microsoft® DirectMusic® application programming interface (API) in Microsoft® DirectX® for C++.

The following samples are found on the **Start** menu under **Programs/Microsoft DirectX 9.0 SDK/C++ Samples/DirectMusic Samples**.

- <u>3D Audio Sample</u>
- <u>Audio Path Sample</u>
- <u>Audio Scripts Sample</u>
- <u>AudioFX Sample</u>
- <u>Cross Fade Sample</u>
- <u>Music Tool Sample</u>
- <u>Play Audio Sample</u>
- <u>Play Motif Sample</u>
- <u>Play Multi Sample</u>

The source code is in the following folder, where *SDK root* is the installation folder for the DirectX SDK, such as C:\DXSDK:

(SDK root)\samples\C++\DirectMusic

In addition to these samples, the source files for tutorial applications are contained in the following folder:

(SDK root)\samples\C++\DirectMusic\Tutorials

For more information, see <u>DirectMusic C++ Tutorials</u>.

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 digital media files. For more information, see <u>DirectMusic Sample Framework</u>.

```
© 2004 Microsoft Corporation. All rights reserved.
```

DirectMusic Sample Framework

The core functionality for the DirectMusic samples is in a common set of classes, declared in Dmutil.h and implemented in Dmutil.cpp. In addition, some of the samples make use of the DirectSound classes declared in Dsutil.h and implemented in Dsutil.cpp.

These files are found in the following folders:

```
(SDK root)\samples\C++\Common\Include
(SDK root)\samples\C++\Common\Src
```

To use the sample framework, your project must link to Dxerr9.lib and Winmm.lib

The framework classes provide basic functionality for the samples, and you can use them as a starting-point for your own applications. However, they are not intended to be a full-featured wrapper for the DirectMusic API.

The following classes are used in the samples:

Class	Description
CMusicManager Sample Class	Contains methods for loading and creating objects, creating the performance, and managing memory.
CMusicScript Sample Class	Represents a script, and contains methods for getting and setting variables and calling routines.
C3DMusicSegment Sample Class	Represents a segment playing on a 3-D audiopath. Inherits from CMusicSegment .
CMusicSegment Sample Class	Represents a segment, and contains methods for downloading and unloading instruments, playing and stopping the segment, and retrieving a style from the segment.

See Also

• <u>DirectMusic C++ Samples</u>

CMusicManager Sample Class

The **CMusicManager** sample class contains methods for loading and creating objects, creating the performance, and managing memory.

```
class CMusicManager
{
protected:
    BOOL
                              m bCleanupCOM;
    IDirectMusicLoader8*
                              m pLoader;
    IDirectMusicPerformance8* m_pPerformance;
    IDirectSound3DListener*
                              m pDSListener;
    DS3DLISTENER
                              m_dsListenerParams;
public:
    CMusicManager();
    ~CMusicManager();
    inline IDirectMusicLoader8* GetLoader()
        { return m_pLoader; }
    inline IDirectMusicPerformance8* GetPerformance()
        { return m_pPerformance; }
    inline IDirectSound3DListener*
                                     GetListener()
        { return m pDSListener; }
    IDirectMusicAudioPath8* GetDefaultAudioPath();
    HRESULT Initialize(HWND hWnd, DWORD dwPChannels = 128,
        DWORD dwDefaultPathType = DMUS_APATH_DYNAMIC_STEREO,
        LPDIRECTSOUND pDS = NULL);
    HRESULT SetSearchDirectory(const TCHAR* strMediaPath);
    VOID
            CollectGarbage();
    VOID
            StopAll();
    HRESULT CreateSegmentFromFile(CMusicSegment** ppSegment,
        TCHAR* strFileName, BOOL bDownloadNow = TRUE,
        BOOL bIsMidiFile = FALSE);
    HRESULT Create3DSegmentFromFile(C3DMusicSegment** ppSegment,
        TCHAR* strFileName, BOOL bDownloadNow = TRUE,
        BOOL bIsMidiFile = FALSE, IDirectMusicAudioPath8* p3DAudioPa
    HRESULT CreateScriptFromFile(CMusicScript** ppScript,
        TCHAR* strFileName);
    HRESULT CreateChordMapFromFile(IDirectMusicChordMap8** ppChordMa
        TCHAR* strFileName);
    HRESULT CreateStyleFromFile(IDirectMusicStyle8** ppStyle,
        TCHAR* strFileName);
    HRESULT GetMotifFromStyle(IDirectMusicSegment8** ppMotif,
        TCHAR* strStyle, TCHAR* wstrMotif);
```

```
HRESULT CreateSegmentFromResource(CMusicSegment** ppSegment,
        TCHAR* strResource, TCHAR* strResourceType,
        BOOL bDownloadNow = TRUE, BOOL bIsMidiFile = FALSE);
        VOID Set3DParameters(FLOAT fDistanceFactor, FLOAT fDopplerFactor
};
```

Constructor

The constructor initializes COM and the private data members.

Public Methods

The class contains the following methods, in alphabetical order.

Method	Description
CollectGarbage	Calls the IDirectMusicLoader8::CollectGarbage method, after ensuring that the loader still exists.
Create3DSegmentFromFile	Loads an object from a file and encapsulates it in a C3DMusicSegment object. If instructed to do so, sets the standard MIDI file parameter and downloads instruments.
CreateChordMapFromFile	Loads a chordmap from a file and returns the IDirectMusicChordMap8 interface.
CreateScriptFromFile	Loads a script from a file and encapsulates it in a CMusicScript object.
CreateSegmentFromFile	Loads an object from a file and encapsulates it in a CMusicSegment object. If instructed to do so, sets the standard MIDI file parameter and downloads instruments.
CreateSegmentFromResource	Loads an object from memory and encapsulates it in a CMusicSegment object. If instructed to do so, sets the standard MIDI file parameter and downloads instruments.
CreateStyleFromFile	Loads a style from a file and returns the IDirectMusicStyle8 interface.
	Returns a pointer to the IDirectMusicAudioPath8 interface for the

GetDefaultAudioPath	default audiopath created in the Initialize method.
GetListener	Returns a pointer to the interface.
GetLoader	Returns a pointer to the IDirectMusicLoader8 interface obtained in the Initialize method.
GetMotifFromStyle	Loads a motif from a style file and returns the IDirectMusicSegment8 interface.
GetPerformance	Returns a pointer to the IDirectMusicPerformance8 interface obtained in the Initialize method.
Initialize	Creates the loader and the performance, and initializes the performance with the supplied parameters. Note that in most cases <i>pDs</i> can be NULL.
Set3DParameters	Sets the distance factor, Doppler factor, and rolloff factor.
SetSearchDirectory	Sets the search directory for all types of objects.
StopAll	Stops all playing segments by calling IDirectMusicPerformance8::Stop , after ensuring that the performance still exists.

The class is implemented in *(SDK root)*\samples\C++\Common\Src\Dmutil.cpp.

See Also

• DirectMusic Sample Framework

CMusicScript Sample Class

The **CMusicScript** sample class represents a script, and contains methods for getting and setting variables and calling routines.

```
class CMusicScript
Ł
protected:
    IDirectMusicScript8*
IDirectMusicLoader8*
                                m_pScript;
                                m pLoader;
    IDirectMusicPerformance8* m_pPerformance;
public:
    CMusicScript(IDirectMusicPerformance8* pPerformance,
                   IDirectMusicLoader8* pLoader,
IDirectMusicScript8* pScript);
    virtual ~CMusicScript();
    inline IDirectMusicScript8* GetScript() { return m_pScript; }
    HRESULT CallRoutine(TCHAR* strRoutine);
    HRESULT SetVariableNumber(TCHAR* strVariable, LONG lValue);
    HRESULT GetVariableNumber(TCHAR* strVariable, LONG* plValue);
    HRESULT SetVariableObject(TCHAR* strVariable, IUnknown *punkValu
    HRESULT GetVariableObject(TCHAR* strVariable, REFIID riid,
        LPVOID FAR *ppv);
};
```

Constructor

The constructor stores pointers to the loader, performance, and script objects.

Public Methods

The class contains the following public methods, in alphabetical order.

Method	Description
CallRoutine	Calls IDirectMusicScript8::CallRoutine, after
	doing any necessary string type conversion.
GetScript	Returns the IDirectMusicScript8 interface pointer.
	Calls
GetVariableNumber	IDirectMusicScript8::GetVariableNumber, after

	doing any necessary string type conversion.
GetVariableObject	Calls IDirectMusicScript8::GetVariableObject,
	after doing any necessary string type conversion.
SetVariableNumber	Calls IDirectMusicScript8::SetVariableNumber,
	after doing any necessary string type conversion.
SetVariableObject	Calls IDirectMusicScript8::SetVariableObject,
	after doing any necessary string type conversion.

The class is implemented in *(SDK root)*\samples\C++\Common\Src\Dmutil.cpp.

See Also

• <u>DirectMusic Sample Framework</u>

C3DMusicSegment Sample Class

The **C3DMusicSegment** sample class represents a segment playing on a 3-D audiopath.

```
class C3DMusicSegment : public CMusicSegment
{
protected:
    IDirectMusicAudioPath8* m p3DAudioPath;
                            m pDS3DBuffer;
    IDirectSound3DBuffer*
    DS3DBUFFER
                            m dsBufferParams;
                            m bDeferSettings;
    BOOL
    BOOL
                            m_bCleanupAudioPath;
public:
    C3DMusicSegment(IDirectMusicPerformance8* pPerformance,
                   IDirectMusicLoader8* pLoader,
                   IDirectMusicSegment8* pSegment,
                   IDirectMusicAudioPath8* pAudioPath);
    virtual ~C3DMusicSegment();
    HRESULT Init();
    IDirectMusicAudioPath8* GetAudioPath() { return m_p3DAudioPath;
    HRESULT Play(DWORD dwFlags = DMUS_SEGF_SECONDARY,
            IDirectMusicAudioPath8* pAudioPath = NULL);
    VOID Set3DParameters(FLOAT fMinDistance, FLOAT fMaxDistance);
    VOID SetObjectProperties(D3DVECTOR* pvPosition, D3DVECTOR* pvVel
};
```

Constructor

The constructor stores pointers to the performance, loader, segment, and audiopath objects.

Public Methods

In addition to the methods inherited from **CMusicSegment**, the class contains the following public methods, in alphabetical order.

Method	Description
GetAudioPath	Returns a pointer to the IDirectMusicAudioPath8 interface in the m_p3DAudioPath member.

Init	Creates a 3-D audiopath if one is not already associated with the segment, and retrieves the 3-D buffer in the m_pDS3DBuffer member.
Play	Plays the segment, using the specified flags and audiopath. If no audiopath is specified, the audiopath in the m_p3DAudioPath member is used.
Set3DParameters	Sets the minimum and maximum distances for the 3-D buffer.
SetObjectProperties	Sets the position and velocity of the 3-D buffer.

The class is implemented in *(SDK root)*\samples\C++\Common\Src\Dmutil.cpp.

See Also

<u>DirectMusic Sample Framework</u>

CMusicSegment Sample Class

The **CMusicSegment** sample class represents a segment, and contains methods for downloading and unloading instruments, playing and stopping the segment, and retrieving a style from the segment.

```
class CMusicSegment
Ł
protected:
    IDirectMusicSegment8*
                              m_pSegment;
    IDirectMusicLoader8*
                              m pLoader;
    IDirectMusicPerformance8* m_pPerformance;
    IDirectMusicAudioPath8*
                              m pEmbeddedAudioPath;
                              m bDownloaded;
    BOOL
public:
    CMusicSegment(IDirectMusicPerformance8* pPerformance,
                  IDirectMusicLoader8* pLoader,
                  IDirectMusicSegment8* pSegment);
    virtual ~CMusicSegment();
    inline IDirectMusicSegment8* GetSegment() { return m_pSegment;
    HRESULT GetStyle(IDirectMusicStyle8** ppStyle,
        DWORD dwStyleIndex = 0);
    HRESULT SetRepeats(DWORD dwRepeats);
    HRESULT Play(DWORD dwFlags = DMUS_SEGF_SECONDARY,
        IDirectMusicAudioPath8* pAudioPath = NULL);
    HRESULT Stop(DWORD dwFlags = 0);
    HRESULT Download( IDirectMusicAudioPath8* pAudioPath = NULL );
    HRESULT Unload(IDirectMusicAudioPath8* pAudioPath = NULL);
    BOOL
            IsPlaying();
};
```

Constructor

The constructor stores pointers to the performance, loader, and segment objects. It also attempts to create an audiopath from an audiopath configuration embedded in the segment.

Public Methods

The class contains the following public methods, in alphabetical order.

Method	Description
Download	Downloads the segment's instruments to the supplied audiopath. If no audiopath is supplied, this method downloads to the embedded audiopath if there is one, or to the performance otherwise.
GetSegment	Retrieves the IDirectMusicSegment8 interface pointer.
GetStyle	Retrieves a style in the segment, if the segment has a style track.
IsPlaying	Returns a Boolean variable that specifies whether the segment is playing. This method calls IDirectMusicPerformance8::IsPlaying .
Play	Plays the segment, using the specified flags and audiopath. If no audiopath is specified, this method plays on the embedded audiopath if there is one, or on the default audiopath otherwise. The method fails if the instruments have not been downloaded.
SetRepeats	Calls IDirectMusicSegment8::SetRepeats.
Unload	Unloads instruments from the specified audiopath. If no audiopath is specified, this method unloads from the embedded audiopath if there is one, or from the performance otherwise.

The class is implemented in *(SDK root)*\samples\C++\Common\Src\Dmutil.cpp.

See Also

• <u>DirectMusic Sample Framework</u>

3D Audio Sample

The 3D Audio 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)\Samples\C++\DirectMusic\3DAudio

Executable: (SDK root)\Samples\C++\DirectMusic\Bin

User's Guide

Click **Open File** and load a WAV, 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

The 3D Audio sample is very similar in form to the <u>Play Audio Sample</u>.

See Also

- <u>DirectMusic C++ Samples</u>
- <u>Retrieving Objects from an Audiopath</u>

Audio Path Sample

The Audio Path 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)\Samples\C++\DirectMusic\AudioPath

Executable: (SDK root)\Samples\C++\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 <u>Play Audio Sample</u>. The Audio Path sample 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. The created **IDirectMusicAudioPath8** interface is pointed to by *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 <u>IDirectMusicPerformance8::PlaySegmentEx</u> as the *pFrom* parameter. The alarm sound is then played as a new primary segment.

See Also

- <u>DirectMusic C++ Samples</u>
- <u>Using Audiopaths</u>

Audio Scripts Sample

The Audio Scripts 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 waveforms can be played as variations in a segment.

Path

Source: (SDK root)\Samples\C++\DirectMusic\AudioScripts

Executable: (SDK root)\Samples\C++\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 to 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.

See Also

- <u>DirectMusic C++ Samples</u>
- <u>Using Audio Scripts</u>

AudioFX Sample

The AudioFX sample shows how to use <u>DMOs</u> on DirectMusic audiopaths to add effects to sounds, and how to set effect parameters.

Path

Source: (SDK root)\Samples\C++\DirectMusic\AudioFX

Executable: (SDK root)\Samples\C++\DirectMusic\Bin

User's Guide

A default sound file is loaded when the application is run. You can 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

• <u>DirectMusic C++ Samples</u>

Cross Fade Sample

The Cross Fade sample shows how to fade out one segment while fading in another.

Path

Source: (SDK root)\Samples\C++\DirectMusic\CrossFade

Executable: *(SDK root)*\Samples\C++\DirectMusic\Bin

User's Guide

Click **Play** to play Segment 1. Click **Crossfade** to fade out Segment 1 and fade in Segment 2. Click **Crossfade** again to fade out Segment 2 and fade in Segment 1. You can load different segments by using the **Browse** buttons.

Programming Notes

The fade is achieved by playing the two segments on different audiopaths, whose volumes can be controlled separately by sending **DMUS_CURVE_PMSG** messages. The **IDirectMusicAudioPath8::SetVolume** method could be used instead, but it does not give any control over scheduling the curve. By using a curve message, the sample can put the starting point of the fade slightly in the future and make sure fading of both audiopaths happens at exactly the same time.

See Also

- <u>DirectMusic C++ Samples</u>
- <u>Curves</u>
- <u>Using Audiopaths</u>

Music Tool Sample

The Music Tool sample demonstrates how to implement a DirectMusic tool that intercepts messages.

Path

Source: (SDK root)\Samples\C++\DirectMusic\MusicTool

Executable: (SDK root)\Samples\C++\DirectMusic\Bin

User's Guide

Play the default segment, or choose another WAV, MIDI, or DirectMusic segment file by clicking **Open File**. Select a tool from the drop-down list. The Echo Tool adds an echo to the sound. The Measure Tool causes the square to the right of the drop-down list to flash green on every beat, and red on every measure boundary.

Programming Notes

The tools are implemented in Echotool.cpp and Meastool.cpp. The Echo Tool works by copying messages and sending the copies to a different channel group. The Measure Tool responds to notifications of type GUID_NOTIFICATION_MEASUREANDBEAT. Notifications don't have to be intercepted by tools; they can also be retrieved by using **IDirectMusicPerformance8::GetNotificationPMsg**, as in the <u>Play Audio Sample</u>.

See Also

- <u>DirectMusic C++ Samples</u>
- <u>DirectMusic Tools</u>
- Notification and Event Handling

Play Audio Sample

The Play Audio 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)\Samples\C++\DirectMusic\PlayAudio

Executable: (SDK root)\Samples\C++\DirectMusic\Bin

User's Guide

Play the default segment, or load another WAV, MIDI, or DirectMusic segment file by clicking **Open File**. Adjust the tempo and volume by using the sliders. The tempo cannot be adjusted for WAV files.

Programming Notes

The **OnInitDialog** function does the following:

- 1. Creates an 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 **<u>IDirectMusicLoader8</u>** by using **CoCreateInstance**.
 - Creates **IDirectMusicPerformance8** by using **CoCreateInstance**.
 - Calls **IDirectMusicPerformance8::InitAudio** to initialize the performance and create a standard audiopath.
- 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 event whose handle is in *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 <u>Garbage Collection</u>.
- 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.
- Calls CMusicManager::CreateSegmentFromFile to create a CMusicSegment called *g_pMusicSegment*. This entails the following steps:
 - Call <u>IDirectMusicLoader8::LoadObjectFromFile</u> to load the <u>IDirectMusicSegment8</u> into *pSegment*.
 - Create a **CMusicSegment**, passing in *pSegment*.
 - If the file is a pure MIDI file, call
 <u>IDirectMusicSegment8::SetParam</u>, 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 IDirectMusicSegment8. The application then compares this pointer 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.

See Also

- <u>DirectMusic C++ Samples</u>
- © 2004 Microsoft Corporation. All rights reserved.

Play Motif Sample

The Play Motif 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)\Samples\C++\DirectMusic\PlayMotif

Executable: *(SDK root)*\Samples\C++\DirectMusic\Bin

User's Guide

Play the default segment, or load another DirectMusic segment based on a <u>style</u> 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 Play Motif sample is very similar in form to the <u>Play Audio Sample</u>.

When loading the file, Play Motif performs the same steps as Play Audio, with the following additions in the **LoadSegmentFile** function:

- Loops through styles in the segment, searching it for motifs. It calls the CMusicSegment::GetStyle method in the sample framework, which in turn calls <u>IDirectMusicSegment8::GetParam</u>, 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, 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.
- 3. 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 and any alignment option.

DirectMusic notifications are handled much as in Play Audio, 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

- <u>DirectMusic C++ Samples</u>
- <u>Using Motifs</u>

Play Multi Sample

The Play Multi sample shows how to play multiple segments simultaneously, one as a primary segment and the others as secondary segments.

Path

Source: (SDK root)\Samples\C++\DirectMusic\PlayMulti

Executable: *(SDK root)*\Samples\C++\DirectMusic\Bin

User's Guide

Use the default segments, or choose others by using the Browse buttons. The primary segment should be style-based so that it contains the rhythmic information necessary for proper cuing of the secondary segments.

Play the primary segment. Select a play boundary for a secondary segment and play it. Notice that the secondary segment begins playing on the specified boundary.

Select **Controlling Segment** to make a secondary segment the control segment. This will have an effect only if the secondary segment has a mute track, command track, tempo track, or chord track. Try loading a secondary segment that has a different tempo than the primary segment and playing it as the control segment. The tempo of the primary segment changes to match that of the secondary segment.

Programming Notes

The flags for **IDirectMusicPerformance8::PlaySegmentEx** are set in the **OnPlay** function and passed to the **CMusicSegment::Play** method in the sample framework.

See Also

- <u>DirectMusic C++ Samples</u>
- <u>Control Segments</u>

• Using Segments

DirectMusic C++ Tutorials

This section contains the following tutorials showing how to implement Microsoft® DirectMusic® in a C++ application:

- Tutorial 1: Playing Audio Files
- Tutorial 2: Using Audiopath Objects

Other, more specialized uses of DirectMusic are demonstrated in the sample applications provided with the SDK. For a description of these samples, see <u>DirectMusic C++ Samples</u>.

Note For the sake of simplicity, the tutorial applications perform minimal error checking when calling DirectX methods. Except for methods that always succeed, your code should always check the result of method calls and handle failure appropriately.

Tutorial 1: Playing Audio Files

This tutorial is a step-by-step guide to the most basic tasks in DirectMusic: initializing a DirectMusic performance and playing an audio file. The tutorial is presented in the following steps:

- <u>Step 1: Initialize</u>
- <u>Step 2: Load a File</u>
- Step 3: Play the File
- <u>Step 4: Close Down</u>

The complete sample code for the tutorial is available in the following folder:

C:\DXSDK\Samples\C++\DirectMusic\Tutorials\Tutorial1

Note If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "Dxsdk" in the path.

Step 1: Initialize

The following instructions are needed for any application that uses the DirectMusic API. Including Dmusici.h also causes 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 <u>performance channels</u> to the audiopath. WAV 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: <u>Step 2: Load a File</u>

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[512];
if( GetWindowsDirectory( strPath, MAX_PATH+1 ) == 0 )
       return 0;
strcat( strPath, "\\media" );
// Convert to Unicode.
WCHAR wstrSearchPath[MAX_PATH + 1];
MultiByteToWideChar( CP_ACP, 0, strPath, -1,
       wstrSearchPath, MAX_PATH );
wstrSearchPath[MAX_PATH] = 0;
// Set the search directory.
q_pLoader->SetSearchDirectory(
  GUID_DirectMusicAllTypes, // Types of files sought.
                             // Where to look.
  wstrSearchPath,
                             // Don't clear object data.
  FALSE
);
```

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"ding.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.",
        "DirectMusic Tutorial", MB_OK );
  g_pPerformance->CloseDown();
  g_pLoader->Release();
  g_pPerformance->Release();
  CoUninitialize();
  return 0;
}
```

Next: Step 3: Play the File

Step 3: Play the File

The WAV 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, // Not used.
    NULL, // For transitions.
    0, // Flags.
    0, // Start time; 0 is immediate.
    NULL, // Pointer that receives segment state.
    NULL, // Object to stop.
    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 five main steps:

- 1. Stop any playing segments by calling **<u>IDirectMusicPerformance8::Stop</u>**.
- 2. Unload any segments that were downloaded to the synthesizer.
- 3. Close down the performance. The <u>IDirectMusicPerformance8::CloseDown</u> method performs miscellaneous cleanup tasks and releases internal references to objects.
- 4. Release all interfaces.
- 5. Close COM.

}

The following code from the **WinMain** function in the tutorial sample is called when the dialog box is closed.

```
q_pPerformance->Stop(
    NULL, // Stop all segments.
   NULL, // Stop all segment states.
         // Do it immediately.
    0,
          // Flags.
    0
);
g_pSegment->Unload(g_pPerformance);
g_pPerformance->CloseDown();
g_pLoader->Release();
q_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:

- <u>Step 1: Create the Audiopath</u>
- <u>Step 2: Retrieve the Buffer</u>
- <u>Step 3: Change Buffer Parameters</u>

The complete sample code for the tutorial is available in the following folder:

```
C:\DXSDK\Samples\C++\DirectMusic\Tutorials\Tutorial2
```

Note If you installed the DirectX SDK in a different root directory, substitute the name of that directory for "Dxsdk" 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 <u>Tutorial 1: Playing Audio Files</u>.

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:

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->	Play	/SegmentEx(
g_pSegment,	//	Segment to play.
NULL,	//	Not used.
NULL,	//	For transitions.
Θ,	//	Flags.
Θ,		Start time; 0 is immediate.
NULL,	//	Pointer that receives segment state.
NULL,	11	Object to stop.

p3DAudioPath // Audiopath.

);

Next: <u>Step 2: Retrieve the Buffer</u>

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 <u>Retrieving Objects from an Audiopath</u>.

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.

The last parameter of the **IDirectSound3DBuffer8::SetPosition** method specifies whether the change is to be made immediately or deferred until all changes are committed.

^{© 2004} Microsoft Corporation. All rights reserved.

DirectMusic C/C++ Reference

This section contains reference information for the API elements of Microsoft® DirectMusic®. Reference material is divided into the following categories.

- <u>DirectMusic Interfaces</u>
- <u>DirectMusic Messages</u>
- <u>DirectMusic Structures</u>
- DLS Structures
- <u>DirectMusic File Format</u>
- <u>DirectMusic File Structures</u>
- <u>Standard Track Parameters</u>
- <u>DirectMusic Enumerated Types</u>
- <u>DirectMusic Return Values</u>

DirectMusic Interfaces

This section contains references for DirectMusic COM interfaces.

Interfaces in the current version (other than **IKsControl** and **IReferenceClock**) 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 interface pointer.

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.

For information on which methods are supported by earlier versions of an interface, see the declaration of the interface in the appropriate header file.

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 Provides methods for managing buffer	<u>'ר</u>
	з,
the master clock.	
IDirectMusicAudioPath8 Manages the stages of data flow from	th
performance to the final mixer.	
IDirectMusicBand8Represents a DirectMusic band object	•
Represents a buffer containing time-st	
IDirectMusicBuffer8 (typically in the form of MIDI messag	es
sequenced to a port.	
IDirectMusicChordMap8 Represents a <u>chordmap</u> .	
IDirectMusicCollection8 Manages an instance of a DLS file.	
IDirectMusicComposer8 Enables access to the composition eng	in
Provides access to objects in a contain	er
IDirectMusicContainer8 collection of objects used by a segment	lt
performance.	
IDirectMusicDownload8 Represents a contiguous memory chur	
downloading to a DLS synthesizer por	
Used to identify an instrument that has	
downloaded to the synthesizer by usin	-
IDirectMusicDownloadedInstrument8 IDirectMusicPort8::DownloadInstrument8	
IDirectMusicPerformance8::Downlo method.	<u>)</u>
IDirectMusicGetLoader8Used by an object parsing a stream whIDirectMusicGetLoader8needs to load another object referenced	
stream.	
IDirectMusicGraph8Manages the loading and message flow	 N
Represents an individual instrument fr	
IDirectMusicInstrument8 collection.	01
IDirectMusicLoader8 Used for finding, enumerating, cachin	g,
objects.	
Used by an object parsing a stream wh	le
IDirectMusicObject8 needs to load another object reference	1 I
stream.	
IDirectMusicPatternTrack8 Represents a track that contains a sing	le

IDirectMusicPerformance8	Manages playback.
IDirectMusicPort8	Represents a device that sends or received data.
IDirectMusicPortDownload8	Enables an application to communicate a port that supports DLS downloading a download memory chunks directly to th
IDirectMusicScript8	Represents a script containing variables set and retrieved by the application, and can be called by the application.
IDirectMusicSegment8	Represents a segment, which is a playab data made up of multiple tracks.
IDirectMusicSegmentState8	Represents a playing instance of a segm
IDirectMusicStyle8	Represents a <u>style</u> object, which encapsı collection of patterns, motifs, and bands time composition of musical segments.
IDirectMusicSynth	Implemented by synthesizers.
IDirectMusicSynthSink	Implemented by synthesizer sinks.
IDirectMusicThru8	Supports thruing of MIDI messages fror port to another port.
IDirectMusicTool8	Represents a tool object that processes n
IDirectMusicTrack8	Represents a track object, which can sto of data for a segment.
IKsControl	Used to get, set, or query the support of events, and methods.
IReferenceClock	Represents a system reference clock.

IDirectMusic8 Interface

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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusic8** interface exposes the following methods.

Method	Description
<u>Activate</u>	Activates or deactivates all ports created from this interface.
<u>CreatePort</u>	Creates an object for a DirectMusic port.
<u>EnumPort</u>	Enumerates and retrieves the capabilities of the DirectMusic ports connected to the system.
<u>GetDefaultPort</u>	Retrieves the GUID of the default output port.

Ports

Timing

Method	Description
EnumMasterClock	Enumerates the clocks that DirectMusic
	can use as the master clock.

<u>GetMasterClock</u>	Retrieves the GUID and a pointer to the interface for the current master clock.
SetExternalMasterClock	Sets the DirectMusic master clock to an existing clock object.
<u>SetMasterClock</u>	Sets the DirectMusic master clock to a clock identified by a call to EnumMasterClock .

Miscellaneous

Method	Description
CreateMusicBuffer	Creates an object to hold messages being sequenced to the port.
<u>SetDirectSound</u>	Connects DirectMusic to a DirectSound device object for output from the synthesizer.

The **LPDIRECTMUSIC8** type is defined as a pointer to the **IDirectMusic8** interface:

typedef IDirectMusic8 *LPDIRECTMUSIC8;

Requirements

Header: Declared in dmusicc.h.

See Also

• <u>DirectMusic Interfaces</u>

IDirectMusic8::Activate

The **Activate** method activates or deactivates all ports created from this interface.

Syntax

```
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 **Activate**(FALSE) when they lose input focus if they do not need to play sounds 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

- IDirectMusic8 Interface
 IDirectMusicPort8::Activate

IDirectMusic8::CreateMusicBuffer

The **CreateMusicBuffer** method creates a **DirectMusicBuffer** object to hold 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.

Syntax

```
HRESULT CreateMusicBuffer(
   LPDMUS_BUFFERDESC pBufferDesc,
   LPDIRECTMUSICBUFFER*..ppBuffer,
   LPUNKNOWN pUnkOuter
);
```

Parameters

pBufferDesc

Address of the **DMUS_BUFFERDESC** structure that contains the description of the 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 error values shown in the following table.

Return code

CLASS_E_NOAGGREGATION E_INVALIDARG E_NOINTERFACE E_OUTOFMEMORY E_POINTER

Requirements

Header: Declared in dmusicc.h.

See Also

• **IDirectMusic8** Interface

IDirectMusic8::CreatePort

The **CreatePort** method creates an object for a DirectMusic port.

Syntax

```
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 to **sizeof(DMUS_PORTPARAMS8)** 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 <u>S_FALSE</u> if a requested parameter is not available.

If it fails, the method can return one of error values in the following table.

Return code
CLASS_E_NOAGGREGATION
DMUS_E_DSOUND_NOT_SET
E_INVALIDARG
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.

Requirements

Header: Declared in dmusicc.h.

See Also

- **<u>IDirectMusic8</u>** Interface
- <u>Using DirectMusic Ports</u>

IDirectMusic8::EnumMasterClock

The **EnumMasterClock** method enumerates the clocks that DirectMusic can use as the master clock. Each time it is called, this method retrieves information about a single clock.

Syntax

```
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 <u>S_FALSE</u> if there is no clock with that index number.

If it fails, the method can return one of the error values shown in the following table.

Return code 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** Interface
- <u>IDirectMusic8::GetMasterClock</u>
- <u>IDirectMusic8::SetMasterClock</u>
- <u>Master Clock</u>

IDirectMusic8::EnumPort

The **EnumPort** method 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.

Syntax

```
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 <u>S_FALSE</u> if there is no port with that index value.

If it fails, the method can return one of the error values shown in the following table.

Return code E_INVALIDARG

E_NOINTERFACE E_POINTER

Remarks

Applications should not rely on or store the index number of a port. Restarting the system or adding or removing ports could cause the index number of a port to change.

Requirements

Header: Declared in dmusicc.h.

See Also

- **IDirectMusic8** Interface
- <u>Using DirectMusic Ports</u>

IDirectMusic8::GetDefaultPort

The **GetDefaultPort** method retrieves the GUID of the default output port. This is the port to be created if GUID_NULL is passed to **IDirectMusic8::CreatePort**.

Syntax

```
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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmusicc.h.

See Also

- <u>Default Port</u>
- **IDirectMusic8** Interface

IDirectMusic8::GetMasterClock

The **GetMasterClock** method retrieves the GUID and a pointer to the **IReferenceClock** interface for the clock that is currently set as the DirectMusic master clock.

Syntax

```
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 error values shown in the following table.

Return code <u>E_NOINTERFACE</u> <u>E_POINTER</u>

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 Interface
- <u>IDirectMusic8::SetMasterClock</u>
- Master Clock

IDirectMusic8::SetDirectSound

The **SetDirectSound** method connects DirectMusic to a DirectSound device object for output from the synthesizer.

This method is not used by most applications. The DirectSound device object is normally created and connected to the performance by **IDirectMusicPerformance8::InitAudio**.

Syntax

```
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 device 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. level.

hWnd

Window handle to the DirectSound device 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. 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 <u>DMUS_E_DSOUND_ALREADY_SET</u>.

Remarks

The specified DirectSound device object is the one used for rendering audio on all ports. This default can be overridden on a particular port by using the **IDirectMusicPort8::SetDirectSound** method.

Whenever the **IDirectMusic8::SetDirectSound** method is called, any existing DirectSound device object is released.

When *pDirectSound* is NULL, a new DirectSound device object is not created until a port that uses DirectSound is activated, and the DirectSound device object is automatically released when the last port using it is deactivated.

If you created the DirectSound device 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.

Requirements

Header: Declared in dmusicc.h.

See Also

- IDirectMusic8 Interface
- IDirectMusicPerformance8::Init

IDirectMusic8::SetExternalMasterClock

The **SetExternalMasterClock** method sets the DirectMusic master clock to an existing clock object. There is only one master clock for all DirectMusic applications.

Syntax

```
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 <u>DMUS E PORTS OPEN</u>.

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** Interface
- IDirectMusic8::EnumMasterClock

- <u>IDirectMusic8::GetMasterClock</u>
- IDirectMusic8::SetMasterClock
 Master Clock

IDirectMusic8::SetMasterClock

The **SetMasterClock** method sets the DirectMusic master clock to a clock identified by a GUID obtained by using the **IDirectMusic8::EnumMasterClock** method. There is only one master clock for all DirectMusic applications.

Syntax

```
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 <u>DMUS_E_PORTS_OPEN</u>.

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 Interface
- <u>IDirectMusic8::EnumMasterClock</u>
- <u>IDirectMusic8::GetMasterClock</u>
- IDirectMusic8::SetExternalMasterClock
- Master Clock

IDirectMusicAudioPath8 Interface

The **IDirectMusicAudioPath8** interface manages the stages of data flow from the performance to the final mixer. An 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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicAudioPath8** interface exposes the following methods.

Description
Activates or deactivates the audiopath.
Translates between a
<u>performance channel</u> in a segment and
the equivalent channel allocated in the
performance for the audiopath.
Retrieves an interface for an object in
the audiopath.
Sets the audio volume on the audiopath.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Interfaces</u>
 <u>Using Audiopaths</u>

IDirectMusicAudioPath8::Activate

The **Activate** method activates or deactivates the audiopath.

Syntax

```
HRESULT Activate(
   BOOL fActivate
);
```

Parameters

fActivate

Boolean that specifies whether to activate (TRUE) or deactivate (FALSE) the audiopath.

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.

See Also

• IDirectMusicAudioPath8 Interface

IDirectMusicAudioPath8::ConvertPChannel

The **ConvertPChannel** method translates between a <u>performance channel</u> 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.

Syntax

```
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 error values shown in the following table.

Return code DMUS_E_NOT_FOUND E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicAudioPath8 Interface

IDirectMusicAudioPath8::GetObjectInPath

The **GetObjectInPath** method retrieves an interface for an object in the audiopath.

Syntax

```
RESULT GetObjectInPath(
  DWORD dwPChannel,
  DWORD dwStage,
  DWORD dwBuffer,
  REFGUID guidObject,
  DWORD dwIndex,
  REFGUID iidInterface,
  void ** ppObject
);
```

Parameters

dwPChannel

<u>Performance channel</u> to search, or DMUS_PCHANNEL_ALL to search all channels. The first channel is numbered 0. (See Remarks.)

dwStage

Stage in the audiopath. Can be one of the values in the following table.

Value	Description
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 <u>DMO</u> in a buffer.
DMUS_PATH_MIXIN_BUFFER	Get a global <u>mix-in buffer</u> .
	Get a DMO in a global mix-in

DMUS_PATH_MIXIN_BUFFER_DMO 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 <u>primary buffer</u> .

dwBuffer

Index of the buffer (if *dwStage* is DMUS_PATH_BUFFER or DMUS_PATH_MIXIN_BUFFER), or index of the buffer in which the DMO resides (if *dwStage* is DMUS_PATH_BUFFER_DMO or DMUS_PATH_MIXIN_BUFFER_DMO).

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 error values shown in the following table.

Return codeDMUS_E_NOT_FOUNDE_INVALIDARGE_OUTOFMEMORYE_NOINTERFACEE_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

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 example function shows how to enumerate the buffers in an audiopath:

```
void DumpAudioPathBuffers(
    IDirectMusicAudioPath *pDirectMusicAudioPath)
{
  DWORD dwBuffer = 0;
  IDirectSoundBuffer *pDirectSoundBuffer;
  while (S_OK == pDirectMusicAudioPath->GetObjectInPath(
      DMUS_PCHANNEL_ALL, DMUS_PATH_BUFFER, dwBuffer,
      GUID_NULL, 0, IID_IDirectSoundBuffer,
      (void**) &pDirectSoundBuffer))
  {
    // Do something with pDirectSoundBuffer.
    // . . .
    dwBuffer++;
    pDirectSoundBuffer->Release();
  }
}
```

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicAudioPath8 Interface
- IDirectMusicSegmentState8::GetObjectInPath
- Retrieving Objects from an Audiopath

IDirectMusicAudioPath8::SetVolume

The **SetVolume** method sets the audio volume on the audiopath. The volume can be faded in or out.

Syntax

```
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 error values shown in the following table.

Return code DMUS_E_NOT_INIT E_INVALIDARG

Remarks

This method works by sending a volume <u>curve</u> message. Any volume events occurring later, such as a band change, override the volume set by this method. **IDirectMusicAudioPath8::SetVolume** is useful mainly for adjusting currently playing sounds; for example, to fade out before stopping a segment. If you want to make a global change that affects all playback, use one of the following techniques:

- Obtain the buffer object from the audiopath and use **IDirectSoundBuffer8::SetVolume**.
- Obtain the port object from the audiopath and use <u>IKsControl::KsProperty</u> to change the GUID_DMUS_PROP_Volume property set.
- Set the master volume for the performance. See <u>Setting and Retrieving</u> <u>Global Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- Cross Fade Sample
- <u>Curves</u>
- IDirectMusicAudioPath8 Interface
- <u>Playing Sounds on Audiopaths</u>

IDirectMusicBand8 Interface

The **IDirectMusicBand8** interface represents a DirectMusic band object. A band is used to set the instrument choices and mixer settings for a set of <u>performance channels</u>. For an overview, see <u>Using Bands</u>. 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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicBand8** interface exposes the following methods.

Method	Description
<u>CreateSegment</u>	Creates a segment object that can be
	played to dynamically perform the
	volume, pan, transposition, and patch
	change commands in the band.
<u>Download</u>	Downloads the <u>DLS</u> data for
	instruments in the band to a
	performance object.
<u>Unload</u>	Unloads the DLS data for instruments
	in the band previously downloaded.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Interfaces</u>
- Using Bands

IDirectMusicBand8::CreateSegment

The **CreateSegment** method creates a segment object that can be played to dynamically perform the volume, pan, transposition, and patch change commands in the band.

Syntax

```
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 error values shown in the following table.

Return code <u>E_FAIL</u> <u>E_OUTOFMEMORY</u> <u>E_POINTER</u>

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicBand8 Interface

• Making Band Changes Programmatically

IDirectMusicBand8::Download

The **Download** method downloads the <u>DLS</u> data for instruments in the band to a performance object.

Syntax

```
HRESULT Download(
    IDirectMusicPerformance* pPerformance
);
```

Parameters

pPerformance

Performance to which instruments are to be downloaded. The performance manages the mapping of <u>performance channels</u> to DirectMusic ports.

Return Values

If the method succeeds, the return value is S_OK, or <u>DMUS_S_PARTIALDOWNLOAD</u>. (See Remarks.)

If it fails, the method can return one of the error values shown in the following table.

Return code

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 in 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 <u>Debugging DirectMusic</u> <u>Projects</u>.

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

- Downloading and Unloading Bands
- IDirectMusicBand8 Interface
 IDirectMusicBand8::Unload

IDirectMusicBand8::Unload

The **Unload** method unloads the <u>DLS</u> data for instruments in the band previously downloaded by <u>IDirectMusicBand8::Download</u>.

Syntax

```
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 error values shown in the following table.

Return code DMUS E NOT INIT

DMUS_E_NOT_FOUND

Requirements

Header: Declared in dmusici.h.

See Also

- Downloading and Unloading Bands
- IDirectMusicBand8 Interface
- IDirectMusicPort8::UnloadInstrument

IDirectMusicBuffer8 Interface

The **IDirectMusicBuffer8** interface represents a buffer containing time-stamped data (typically in the form of MIDI messages) being sequenced to a port. The buffer contains a small amount of data, typically less than 200 milliseconds. Unless your application is doing its own sequencing, you do not need to use the methods of this interface.

IDirectMusicBuffer8 is a type definition for **IDirectMusicBuffer**. The two interface names are interchangeable.

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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicBuffer8** interface exposes the following methods.

Method	Description
<u>Flush</u>	Discards all data in the buffer.
<u>GetBufferFormat</u>	Retrieves the GUID that represents the buffer format.
<u>GetMaxBytes</u>	Retrieves the number of bytes that can be stored in the buffer.
<u>GetNextEvent</u>	Returns information about the next message in the buffer and advances the read pointer.
<u>GetRawBufferPtr</u>	Returns a pointer to the underlying buffer data structure.
<u>GetStartTime</u>	Retrieves the start time of the data in the buffer, relative to the master clock.
<u>GetUsedBytes</u>	Retrieves the number of bytes of data in the buffer.
	Inserts fixed-length data (typically a MIDI channel message), along with

PackStructured	timing and routing information, into the buffer.
PackUnstructured	Inserts unstructured data (typically a MIDI system-exclusive message), along with timing and routing information, into the buffer
ResetReadPtr	Sets the read pointer to the start of the data in the buffer.
<u>SetStartTime</u>	Sets the start time of the data in the buffer, relative to the master clock.
<u>SetUsedBytes</u>	Sets the number of bytes of data in the buffer.
<u>TotalTime</u>	Returns the total time spanned by the data in the buffer.

The **LPDIRECTMUSICBUFFER8** type is defined as a pointer to the **IDirectMusicBuffer8** interface:

typedef IDirectMusicBuffer8 *LPDIRECTMUSICBUFFER8;

Requirements

Header: Declared in dmusicc.h.

See Also

- DirectMusic Interfaces
- <u>IDirectMusic8::CreateMusicBuffer</u>

IDirectMusicBuffer8::Flush

The **Flush** method discards all data in the buffer.

Syntax

HRESULT Flush();

Parameters

None.

Return Values

The method returns S_OK.

Requirements

Header: Declared in dmusicc.h.

See Also

• **IDirectMusicBuffer8** Interface

IDirectMusicBuffer8::GetBufferFormat

The **GetBufferFormat** method retrieves the GUID that represents the buffer format.

Syntax

```
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 <u>**E**POINTER</u>.

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

- <u>DMUS_EVENTHEADER</u>
- <u>IDirectMusic8::CreateMusicBuffer</u>
- **IDirectMusicBuffer8** Interface

IDirectMusicBuffer8::GetMaxBytes

The **GetMaxBytes** method retrieves the number of bytes that can be stored in the buffer.

Syntax

```
HRESULT GetMaxBytes(
   LPDWORD pcb
);
```

Parameters

pcb

Address of a variable that receives 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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmusicc.h.

See Also

• IDirectMusicBuffer8 Interface

IDirectMusicBuffer8::GetNextEvent

The **GetNextEvent** method returns information about the next message in the buffer and advances the read pointer.

Syntax

```
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 **<u>E_POINTER</u>**.

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** Interface
- <u>IDirectMusicBuffer8::ResetReadPtr</u>

IDirectMusicBuffer8::GetRawBufferPtr

The **GetRawBufferPtr** method returns a pointer to the underlying buffer data structure.

Syntax

```
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 **<u>E_POINTER</u>**.

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.

See Also

• **IDirectMusicBuffer8** Interface

IDirectMusicBuffer8::GetStartTime

The **GetStartTime** method retrieves the start time of the data in the buffer, relative to the master clock.

Syntax

```
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 error values shown in the following table.

Return code

DMUS_E_BUFFER_EMPTY E_POINTER

Requirements

Header: Declared in dmusicc.h.

See Also

- **IDirectMusicBuffer8** Interface
- IDirectMusicBuffer8::SetStartTime
- <u>IDirectMusicBuffer8::TotalTime</u>

IDirectMusicBuffer8::GetUsedBytes

The **GetUsedBytes** method retrieves the number of bytes of data in the buffer.

Syntax

```
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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmusicc.h.

See Also

- **IDirectMusicBuffer8** Interface
- IDirectMusicBuffer8::SetUsedBytes

IDirectMusicBuffer8::PackStructured

The **PackStructured** method inserts fixed-length data (typically a MIDI channel message), along with timing and routing information, into the buffer.

Syntax

```
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 error values shown in the following table.

Return codeDMUS_E_INVALID_EVENTE_OUTOFMEMORY

Remarks

At least 32 bytes (the size of <u>DMUS_EVENTHEADER</u> 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** Interface
- IDirectMusicBuffer8::PackUnstructured

IDirectMusicBuffer8::PackUnstructured

The **PackUnstructured** method inserts unstructured data (typically a MIDI system-exclusive message), along with timing and routing information, into the buffer.

Syntax

```
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 error values shown in the following

table.

Return code <u>E_OUTOFMEMORY</u> <u>E_POINTER</u>

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** Interface
- <u>IDirectMusicBuffer8::PackStructured</u>

IDirectMusicBuffer8::ResetReadPtr

The **ResetReadPtr** method sets the read pointer to the start of the data in the buffer.

Syntax

HRESULT ResetReadPtr()

Parameters

None.

Return Values

The method always returns S_OK.

Requirements

Header: Declared in dmusicc.h.

See Also

- **IDirectMusicBuffer8** Interface
- <u>IDirectMusicBuffer8::GetNextEvent</u>

IDirectMusicBuffer8::SetStartTime

The **SetStartTime** method sets the start time of the data in the buffer, relative to the master clock.

Syntax

```
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** Interface
- <u>IDirectMusicBuffer8::GetStartTime</u>

IDirectMusicBuffer8::SetUsedBytes

The **SetUsedBytes** method sets the number of bytes of data in the buffer.

Syntax

```
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 <u>DMUS E BUFFER FULL</u>.

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 Interface

• IDirectMusicBuffer8::GetUsedBytes

IDirectMusicBuffer8::TotalTime

The **TotalTime** method retrieves the total time spanned by the data in the buffer.

Syntax

```
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 <u>**E**POINTER</u>.

Requirements

Header: Declared in dmusicc.h.

See Also

- **IDirectMusicBuffer8** Interface
- <u>IDirectMusicBuffer8::GetStartTime</u>

IDirectMusicChordMap8 Interface

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 composed at run time. 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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicChordMap8** interface exposes the following method.

Method	Description
<u>GetScale</u>	Retrieves the scale associated with the
	chordmap.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Interfaces</u>
- Using Chordmaps

IDirectMusicChordMap8::GetScale

The **GetScale** method retrieves the scale associated with the <u>chordmap</u>.

Syntax

```
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 <u>**E**</u><u>POINTER</u>.

Remarks

The scale is defined by the bits in a **DWORD**, split into a scale pattern in the lower 24 bits and a root in the 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.

See Also

• IDirectMusicChordMap8 Interface

• <u>Using Chordmaps</u>

IDirectMusicCollection8 Interface

The **IDirectMusicCollection8** interface manages an instance of a <u>DLS</u> file. The collection provides methods to access instruments and download them to the synthesizer by means of the <u>IDirectMusicPort8</u> 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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicCollection8** interface exposes the following methods.

Method	Description
<u>EnumInstrument</u>	Retrieves the patch and name of an
	instrument by its index in the collection.
<u>GetInstrument</u>	Retrieves an instrument by its patch
	number.

Requirements

Header: Declared in dmusicc.h.

See Also

- <u>DirectMusic Interfaces</u>
- <u>Using Instrument Collections</u>

IDirectMusicCollection8::EnumInstrument

The **EnumInstrument** method retrieves the patch number and name of an instrument by its index in the collection.

Syntax

```
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** elements 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 error values shown in the following table.

Return code

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 <u>bank select</u>. For more information, see <u>MIDI Channel Messages</u>.

Although the ordering of the enumeration is consistent within one instance of a <u>DLS</u> 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 <u>Working with Instruments</u>.

Requirements

Header: Declared in dmusicc.h.

See Also

- **IDirectMusicCollection8** Interface
- **Using Instrument Collections**

IDirectMusicCollection8::GetInstrument

The **GetInstrument** method retrieves an instrument by its patch number.

Syntax

```
HRESULT GetInstrument(
   DWORD dwPatch,
   IDirectMusicInstrument** ppInstrument
);
```

Parameters

dwPatch

Instrument patch number.

ppInstrument

Address of a variable that receives a pointer to the **<u>IDirectMusicInstrument8</u>** interface.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the error values shown in the following table.

```
Return codeDMUS_E_INVALIDPATCHE_FAILE_OUTOFMEMORYE_POINTER
```

Remarks

The patch number passed in *dwPatch* describes the full patch address, including the MIDI parameters for MSB and LSB <u>bank select</u>. MSB is shifted left 16 bits, and LSB is shifted left 8 bits. 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 <u>Working with Instruments</u>.

Requirements

Header: Declared in dmusicc.h.

See Also

- **IDirectMusicCollection8** Interface
- <u>MIDI Channel Messages</u>
- Using Instrument Collections

IDirectMusicComposer8 Interface

The **IDirectMusicComposer8** interface enables access to the composition engine. In addition to building new segments from <u>templates</u> and <u>chordmaps</u>, 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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicComposer8** interface exposes the following methods.

Method	Description
<u>AutoTransition</u>	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.
<u>ChangeChordMap</u>	Modifies the chords and scale pattern of
	an existing segment to reflect a new
	chordmap.
<u>ComposeSegmentFromShape</u>	Creates an original segment from a <u>style</u>
	and a <u>chordmap</u> , based on a predefined
	shape.
<u>ComposeSegmentFromTemplate</u>	Creates an original segment from a style,
	a <u>chordmap</u> , and a <u>template</u> .
ComposeTemplateFromShape	Creates a new <u>template</u> segment, based
	on a predefined shape.
ComposeTransition	Composes a transition from a measure
	inside one segment to another.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Interfaces</u>
- <u>Using Compositional Elements</u>

IDirectMusicComposer8::AutoTransition

The **AutoTransition** method 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.

Syntax

```
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 values of the **DMUS_COMMANDT_TYPES** enumeration, or a value defined in DirectMusic Producer as a custom embellishment. If this value is DMUS_COMMANDT_ENDANDINTRO, the method composes a segment containing both an ending to the current primary segment and an introduction to *pToSeg*.

dwFlags

Composition options. See **DMUS_COMPOSEF_FLAGS**.

pChordMap

Pointer to the **<u>IDirectMusicChordMap8</u>** interface of the <u>chordmap</u> 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 error values shown in the following table.

Return code

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, the composition engine attempts 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 Interface
- IDirectMusicComposer8::ComposeTransition
- <u>Using Transitions</u>

IDirectMusicComposer8::ChangeChordMap

The **ChangeChordMap** method modifies the chords and scale pattern of an existing segment to reflect a new <u>chordmap</u>.

Syntax

```
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 <u>style</u>.

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 **<u>E_POINTER</u>**.

Remarks

The method can be called while the segment is playing.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicComposer8 Interface
- <u>Using Chordmaps</u>

IDirectMusicComposer8::ComposeSegmentFromShape

The **ComposeSegmentFromShape** method creates an original segment from a <u>style</u> and a <u>chordmap</u>, based on a predefined shape. The shape represents the way chords and embellishments occur over time across the segment.

Syntax

```
HRESULT ComposeSegmentFromShape(
   IDirectMusicStyle* pStyle,
   WORD wNumMeasures,
   WORD wShape,
   WORD wActivity,
   BOOL fIntro,
   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 one of the error values shown in the following table.

Return code

E_INVALIDARG E_OUTOFMEMORY E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicComposer8 Interface
- IDirectMusicComposer8::ComposeSegmentFromTemplate
- IDirectMusicComposer8::ComposeTemplateFromShape
- <u>Using Compositional Elements</u>

IDirectMusicComposer8::ComposeSegmentFromTemplate

The **ComposeSegmentFromTemplate** method creates an original segment from a <u>style</u>, a <u>chordmap</u>, and a <u>template</u>.

Syntax

```
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 error values shown in the following table.

Return code E_INVALIDARG E_OUTOFMEMORY E_POINTER

Remarks

If *pStyle* is not 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 not 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 Interface
- <u>IDirectMusicComposer8::ComposeSegmentFromShape</u>
- IDirectMusicComposer8::ComposeTemplateFromShape
- <u>Using Templates</u>

IDirectMusicComposer8::ComposeTemplateFromShape

The **ComposeTemplateFromShape** method creates a new <u>template</u> segment, based on a predefined shape.

Syntax

```
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 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 error values shown in the following table.

Return code

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 <u>style</u> 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 patterns from the basic <u>groove level</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicComposer8** Interface
- IDirectMusicComposer8::ComposeSegmentFromTemplate
- <u>Using Templates</u>

IDirectMusicComposer8::ComposeTransition

The **ComposeTransition** method composes a transition from a measure inside one segment to another segment.

Syntax

```
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 a value defined in DirectMusic Producer as a custom embellishment. 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 values from the **DMUS_COMPOSEF_FLAGS** enumerated type.

pChordMap

Pointer to the **<u>IDirectMusicChordMap8</u>** interface of the <u>chordmap</u> 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 error values shown in the following table.

Return code

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 <u>style</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicComposer8 Interface
- IDirectMusicComposer8::AutoTransition
- <u>Using Transitions</u>

IDirectMusicContainer8 Interface

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 <u>DLS</u> collections and streaming waveforms do, the stream also stays open. As a result, the container file stays locked, just as an individual WAV 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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicContainer8** interface exposes the following method.

Method	Description
<u>EnumObject</u>	Retrieves information about an object in
	the container.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>Containers</u>

• <u>DirectMusic Interfaces</u>

IDirectMusicContainer8::EnumObject

The **EnumObject** method retrieves information about an object in the container.

Syntax

```
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.

Return code	Description
S_OK	The object was enumerated.
S_FALSE	There is no object with an index of <i>dwIndex</i> .
DMUS_S_STRING_TRUNCATED	The alias is longer than MAX_PATH.
DMUS_S_GARBAGE_COLLECTED	See <u>Garbage Collection</u> .

If the method fails, it can return **<u>E_POINTER</u>**.

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 <u>Containers</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>Containers</u>
- IDirectMusicContainer8 Interface

IDirectMusicDownload8 Interface

The **IDirectMusicDownload8** interface represents a contiguous memory chunk used for downloading to a <u>DLS</u> 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 <u>Low-Level DLS</u>.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicDownload8** interface exposes the following method.

Method	Description
<u>GetBuffer</u>	Retrieves a pointer to a buffer containing data to be downloaded.

Requirements

Header: Declared in dmusicc.h.

See Also

- DirectMusic Interfaces
- Low-Level DLS

IDirectMusicDownload8::GetBuffer

The **GetBuffer** method retrieves a pointer to a buffer containing data to be downloaded.

Syntax

```
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 error values shown in the following table.

Return codeDMUS_E_BUFFERNOTAVAILABLEE_POINTER

Remarks

The method returns DMUS_E_BUFFERNOTAVAILABLE if the buffer has already been downloaded.

Requirements

Header: Declared in dmusicc.h.

See Also

- IDirectMusicDownload8 Interface
- Low-Level DLS

IDirectMusicDownloadedInstrument8 Interface

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 <u>Working with Instruments</u>.

IDirectMusicDownloadedInstrument8 is a type definition for **IDirectMusicDownloadedInstrument**. The two interface names are interchangeable.

The **IDirectMusicDownloadedInstrument8** interface has no methods other than those inherited from **IUnknown**.

Requirements

Header: Declared in dmusicc.h.

See Also

- DirectMusic Interfaces
- Low-Level DLS

IDirectMusicGetLoader8 Interface

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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicGetLoader8** interface exposes the following method.

Method	Description
<u>GetLoader</u>	Retrieves a pointer to the loader object
	that created the stream.

Requirements

Header: Declared in dmusici.h.

See Also

See Also

- <u>Custom Loading</u>
- IDirectMusicLoader8 Interface
- Loading Audio Data

IDirectMusicGetLoader8::GetLoader

The **GetLoader** method retrieves a pointer to the loader object that created the stream.

Syntax

```
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 <u>E_NOINTERFACE</u>.

Remarks

The following example function finds a reference to an object that needs to be accessed by the loader:

```
HRESULT GetReferencedObject(
 DMUS_OBJECTDESC *pDesc, // Description of object.
 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_IDirectMusicLoader,
      (void**) ppIObject);
    pILoader->Release();
    }
    pIGetLoader->Release();
}
```

Requirements

Header: Declared in dmusici.h.

See Also

- Custom Loading
- **IDirectMusicGetLoader8** Interface

IDirectMusicGraph8 Interface

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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicGraph8** interface exposes the following methods.

Method	Description
<u>GetTool</u>	Retrieves a tool by index.
InsertTool	Inserts a tool in the graph.
RemoveTool	Removes a tool from the graph.
<u>StampPMsg</u>	Stamps a message with a pointer to the next tool that is to receive it.

Requirements

Header: Declared in dmusici.h.

See Also

- DirectMusic Interfaces
- <u>DirectMusic Tools</u>

IDirectMusicGraph8::GetTool

The **GetTool** method retrieves a tool by index.

Syntax

```
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 error values shown in the following table.

Return code DMUS_E_NOT_FOUND E_POINTER

Remarks

The application is responsible for releasing the retrieved tool.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Tools</u>
- **IDirectMusicGraph8** Interface

IDirectMusicGraph8::InsertTool

The **InsertTool** method inserts a tool in the graph.

Note Do not use or distribute tools from untrusted sources. Tools can contain unsafe code.

Syntax

```
HRESULT InsertTool(
    IDirectMusicTool * pTool,
    DWORD * pdwPChannels,
    DWORD cPChannels,
    LONG lIndex
);
```

Parameters

pTool

Tool to insert.

pdwPChannels

Address of an array of <u>performance channels</u> 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 value 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 error values shown in the following table.

Return code

DMUS_E_ALREADY_EXISTS E_OUTOFMEMORY E_POINTER

Remarks

The reference count of the tool is incremented.

This method calls **<u>IDirectMusicTool8::Init</u>**.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Tools</u>
- **IDirectMusicGraph8** Interface

IDirectMusicGraph8::RemoveTool

The **RemoveTool** method removes a tool from the graph.

Syntax

```
HRESULT RemoveTool(
    IDirectMusicTool * pTool
);
```

Parameters

pTool

Tool to remove.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the error values shown in the following table.

Return code DMUS_E_NOT_FOUND E_POINTER

Remarks

The graph's reference to the tool object is released.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Tools</u>
 <u>IDirectMusicGraph8</u> Interface

IDirectMusicGraph8::StampPMsg

The **StampPMsg** method 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.

Syntax

```
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 <u>DMUS_S_LAST_TOOL</u>. (See Remarks.)

If it fails, the method can return **<u>E_POINTER</u>**.

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 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 segment state and performance objects always return S_OK on success. The implementation in the graph 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

- <u>DirectMusic Tools</u>
- IDirectMusicGraph8 Interface

IDirectMusicInstrument8 Interface

The **IDirectMusicInstrument8** interface represents an individual instrument from a <u>DLS</u> collection.

IDirectMusicInstrument8 is a type definition for **IDirectMusicInstrument**. The two interface names are interchangeable.

To create an instrument object, first create a collection object, and then call the **IDirectMusicCollection8::GetInstrument** method. **GetInstrument** creates an instrument object and returns its **IDirectMusicInstrument8** interface pointer.

To download the instrument, pass its interface pointer to the **IDirectMusicPort8::DownloadInstrument** or **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::SetPatch** method.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicInstrument8** interface exposes the following methods.

Method	Description
<u>GetPatch</u>	Retrieves the patch number for the
	instrument.
<u>SetPatch</u>	Sets the patch number for the
	instrument.

The **LPDIRECTMUSICINTRUMENT8** type is defined as a pointer to this interface:

typedef IDirectMusicInstrument8 *LPDIRECTMUSICINSTRUMENT8;

Requirements

Header: Declared in dmusicc.h.

See Also

- <u>DirectMusic Interfaces</u>
- Working with Instruments

IDirectMusicInstrument8::GetPatch

The **GetPatch** method retrieves the patch number for the instrument. The patch number is an address composed of the MSB and LSB <u>bank select</u> and the MIDI patch (program change) number. An optional flag bit indicates that the instrument is a drum, rather than a melodic instrument.

Syntax

```
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 error values shown in the following table.

Return codeDMUS_E_NOT_INITE_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 a drum kit.

Requirements

Header: Declared in dmusicc.h.

See Also

- IDirectMusicInstrument8 Interface
- <u>MIDI Channel Messages</u>
- Working with Instruments

IDirectMusicInstrument8::SetPatch

The **SetPatch** method sets the patch number for the instrument. Although each instrument in a <u>DLS</u> collection has a predefined patch number, the patch number can be reassigned after the <u>IDirectMusicCollection8::GetInstrument</u> method has been used to retrieve the instrument from the collection. For more information on DirectMusic patch numbers, see <u>IDirectMusicInstrument8::GetPatch</u>.

Syntax

```
HRESULT SetPatch(
    DWORD dwPatch
);
```

Parameters

dwPatch

Patch number to assign.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the error values shown in the following table.

Return codeDMUS_E_NOT_INITDMUS_E_INVALIDPATCH

Remarks

The following code example gets an instrument from a collection, remaps its MSB <u>bank select</u> to a different bank, and then downloads the instrument:

```
HRESULT RemappedDownload(
    IDirectMusicCollection8 *pCollection,
    IDirectMusicPort8 *pPort,
    IDirectMusicDownloadedInstrument8 **ppDLInstrument,
    BYTE bMSB,
    DWORD dwPatch)
{
  HRESULT hr;
  IDirectMusicInstrument8* pInstrument;
  hr = pCollection->GetInstrument(dwPatch, &pInstrument);
  if (SUCCEEDED(hr))
  {
    dwPatch &= 0xFF00FFFF;
    dwPatch |= bMSB << 16;
    pInstrument->SetPatch(dwPatch);
    hr = pPort->DownloadInstrument(pInstrument,
                                    ppDLInstrument, NULL, 0);
    pInstrument->Release();
  }
  return hr;
}
```

Requirements

Header: Declared in dmusicc.h.

See Also

- **IDirectMusicInstrument8** Interface
- MIDI Channel Messages
- Working with Instruments

IDirectMusicLoader8 Interface

The **IDirectMusicLoader8** interface is used for finding, enumerating, caching, and loading objects. For an overview, see <u>Loading Audio Data</u>.

This interface supersedes **IDirectMusicLoader** and adds support for garbage collection.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicLoader8** interface exposes the following methods.

Method	Description	
<u>CacheObject</u>	Stores a reference to an object for later use, so that the object is not loaded more than once.	
<u>ClearCache</u>	Removes all saved references to a specified object type.	
<u>CollectGarbage</u>	Removes from the cache objects that are no longer in use.	
EnableCache	Enables or disables automatic caching of all objects loaded.	
EnumObject	Enumerates all available objects of the specified type.	
<u>GetObject</u>	Retrieves an object from a file or resource.	
LoadObjectFromFile	Retrieves an object from a file.	
<u>ReleaseObjectByUnknown</u>	Releases the loader's reference to an object.	
<u>ReleaseObject</u>	Releases the loader's reference to an object.	
<u>ScanDirectory</u>	Searches a directory or disk for all files of a specified class type and file name extension.	
	Enables the loader to find an object when it is later referenced by another	

<u>SetObject</u>	object that is being loaded, and adds attributes to an object so that it can be identified by those attributes.
SetSearchDirectory	Sets a search path for finding object files.

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
Style	CLSID_DirectMusicStyle	sty
Template	CLSID_DirectMusicSegment	tpl
Toolgraph	CLSID_DirectMusicGraph	tgr
Wave	CLSID_DirectSoundWave	wav

Requirements

Header: Declared in dmusici.h.

See Also

- DirectMusic Interfaces
- Loading Audio Data

IDirectMusicLoader8::CacheObject

The **CacheObject** method stores a reference to an object for later use, so that the object is not loaded more than once.

Syntax

```
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 error values shown in the following table.

Return code

E_POINTER DMUS_E_LOADER_OBJECTNOTFOUND

Requirements

Header: Declared in dmusici.h.

See Also

• Cache Management

- IDirectMusicLoader8 Interface
- <u>IDirectMusicLoader8::ClearCache</u>
- <u>IDirectMusicLoader8::EnableCache</u>
- IDirectMusicLoader8::ReleaseObject

IDirectMusicLoader8::ClearCache

The **ClearCache** method removes all saved references to a specified object type.

Syntax

```
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

- <u>Cache Management</u>
- IDirectMusicLoader8 Interface
 IDirectMusicLoader8::CacheObject

IDirectMusicLoader8::CollectGarbage

The **CollectGarbage** method removes from the cache objects that are no longer in use.

Syntax

CollectGarbage();

Parameters

None.

Return Values

None.

Remarks

When an application calls **IDirectMusicLoader8::ReleaseObject**, the 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 Interface

IDirectMusicLoader8::EnableCache

The **EnableCache** method enables or disables automatic caching of all objects it loads. By default, caching is enabled for all classes.

Syntax

```
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 stop caching, or GUID_DirectMusicAllTypes for 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

When you disable caching for a class, all objects of that class that have already been cached are released.

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 <u>styles</u>, <u>chordmaps</u>, and <u>DLS</u> collections, continue to be cached. The first call to **EnableCache** would normally be unnecessary, because caching is enabled for all objects by default.

```
void PrepareLoader(IDirectMusicLoader8 *pILoader)
{
    pILoader->EnableCache(GUID_DirectMusicAllTypes, TRUE);
    pILoader->EnableCache(CLSID_DirectMusicSegment, FALSE);
}
```

Requirements

Header: Declared in dmusici.h.

See Also

- <u>Cache Management</u>
- IDirectMusicLoader8 Interface
- IDirectMusicLoader8::CacheObject
- IDirectMusicLoader8::ClearCache

IDirectMusicLoader8::EnumObject

The **EnumObject** method enumerates an available object of the requested type. Objects are available if they have been loaded or if **IDirectMusicLoader8::ScanDirectory** has been called on the search directory.

Syntax

```
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

Zero-based index into the list of matching objects.

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.

Requirements

Header: Declared in dmusici.h.

See Also

- Enumerating Objects
- **IDirectMusicLoader8** Interface
- IDirectMusicLoader8::ScanDirectory

IDirectMusicLoader8::GetObject

The **GetObject** method retrieves an object from a file or resource and returns the specified interface.

Syntax

```
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 <u>DirectMusic Interface GUIDs</u>.

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 <u>style</u> referenced in a segment. The loader might fail to find the style if it is referenced by name but <u>IDirectMusicLoader8::ScanDirectory</u> 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 error values shown in the following

table.

Return code

DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FORMATNOTSUPPORTED
DMUS_E_LOADER_NOCLASSID
E_FAIL
E_INVALIDARG
E_OUTOFMEMORY
E_POINTER
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. The **dwValidData** flags are evaluated in the following order:

- 1. DMUS_OBJ_OBJECT
- 2. DMUS_OBJ_STREAM
- 3. DMUS_OBJ_MEMORY
- 4. DMUS_OBJ_FILENAME and DMUS_OBJ_FULLPATH
- 5. DMUS_OBJ_NAME and DMUS_OBJ_CATEGORY
- 6. DMUS_OBJ_NAME
- 7. DMUS_OBJ_FILENAME

In other words, the highest priority goes to a unique GUID, followed by a stream pointer, 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.

Do not load data from untrusted sources. Loading DirectMusic data files causes objects to be constructed, with the possibility that excessive demand on resources will lead to degradation of performance or system failure.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicLoader8 Interface
- IDirectMusicLoader8::LoadObjectFromFile
- IDirectMusicLoader8::ReleaseObject
- IDirectMusicLoader8::ScanDirectory
- Loading Audio Data

IDirectMusicLoader8::LoadObjectFromFile

The **LoadObjectFromFile** method retrieves an object from a file and returns the specified interface. This method can be used instead of **IDirectMusicLoader8::GetObject** when the object is in a file.

Syntax

```
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 <u>DirectMusic Interface GUIDs</u>.

pwzFilePath

Name of the file that contains the object. The path can be fully qualified or relative to the search directory. (See Remarks.)

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 error values shown in the following table.

Return code

DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FORMATNOTSUPPORTED
DMUS_E_LOADER_NOCLASSID
E_FAIL
E_INVALIDARG
E_OUTOFMEMORY
E_POINTER
REGDB E CLASSNOTREG

Remarks

Do not load data from untrusted sources. Loading DirectMusic data files causes objects to be constructed, with the possibility that excessive demand on resources will lead to degradation of performance or system failure.

When *pwzFilePath* is an unqualified file name or a relative path, the loader searches first in the current directory, then in the Windows search path, and finally in the directory set by the last call to **IDirectMusicLoader8::SetSearchDirectory**.

DMUS_S_PARTIALLOAD is returned if any referenced object cannot be found, such as a <u>style</u> referenced in a segment. The loader might fail to find the style if it is referenced by name but <u>IDirectMusicLoader8::ScanDirectory</u> 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 Interface
- IDirectMusicLoader8::GetObject

- IDirectMusicLoader8::SetSearchDirectory
 Loading an Object from a File

IDirectMusicLoader8::ReleaseObject

The **ReleaseObject** method releases the loader's reference to an object.

Syntax

```
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 <u>E_POINTER</u>.

Remarks

ReleaseObject is the reciprocal of **<u>IDirectMusicLoader8::CacheObject</u>**.

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

- <u>Cache Management</u>
- IDirectMusicLoader8 Interface
- <u>IDirectMusicLoader8::GetObject</u>
- IDirectMusicLoader8::ReleaseObjectByUnknown

IDirectMusicLoader8::ReleaseObjectByUnknown

The **ReleaseObjectByUnknown** method releases the loader's reference to an object. This method is similar to **IDirectMusicLoader8::ReleaseObject** and is suitable for releasing objects for which the **IDirectMusicObject8** interface is not readily available.

Syntax

```
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 **<u>E_POINTER</u>**.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>Cache Management</u>
- IDirectMusicLoader8 Interface
- <u>IDirectMusicLoader8::GetObject</u>

IDirectMusicLoader8::ScanDirectory

The **ScanDirectory** method searches a directory or disk for all files of a specified 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.

Syntax

```
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

Name of a file to use for 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 error values shown in the following table.

Return code

DMUS_E_NOT_FOUND E_FAIL E_OUTOFMEMORY E_POINTER REGDB_E_CLASSNOTREG

Remarks

The **<u>IDirectMusicLoader8::SetSearchDirectory</u>** 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 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

- **IDirectMusicLoader8** Interface
- Scanning a Directory for Objects

IDirectMusicLoader8::SetObject

The **SetObject** method enables the loader to find an object when it is later referenced by another object that is being loaded, and adds attributes to an object so that it can be identified by those attributes.

Syntax

```
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 error values shown in the following table.

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, the method 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 Interface
- <u>IDirectMusicLoader8::GetObject</u>
- <u>Setting Objects</u>

IDirectMusicLoader8::SetSearchDirectory

The **SetSearchDirectory** method sets a search path for finding object files. The search path can be set for one object file type or for all files.

Syntax

```
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 the loader from accessing objects in a previous directory when those objects 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 error values shown in the following table.

Return code DMUS_E_LOADER_BADPATH E_OUTOFMEMORY E_POINTER

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 Interface
- IDirectMusicLoader8::ScanDirectory
- <u>Setting the Loader's Search Directory</u>

IDirectMusicObject8 Interface

The **IDirectMusicObject8** interface is a generic object interface. All DirectMusic objects that can be loaded from a file or resource support the **IDirectMusicObject8** interface so that they can work with the DirectMusic loader. New types of objects must implement this interface.

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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicObject8** interface exposes the following methods.

Method	Description
<u>GetDescriptor</u>	Retrieves the object's internal
	description.
	Given a file stream, scans the file for
ParseDescriptor	data that it can store in the
	DMUS_OBJECTDESC structure.
<u>SetDescriptor</u>	Sets some or all members of the object's
	internal description.

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

- <u>Custom Loading</u>
- <u>DirectMusic Interfaces</u>
- Loading Audio Data

IDirectMusicObject8::GetDescriptor

The **GetDescriptor** method retrieves the object's internal description.

Syntax

```
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 <u>DMUS_S_GARBAGE_COLLECTED</u>. See <u>Garbage Collection</u>.

If it fails, the method can return <u>E_POINTER</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>Getting Object Descriptors</u>
- IDirectMusicObject8 Interface
- <u>IDirectMusicObject8::SetDescriptor</u>

IDirectMusicObject8::ParseDescriptor

Given a file stream, the **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.

Syntax

```
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 <u>DMUS_S_GARBAGE_COLLECTED</u>. See <u>Garbage Collection</u>.

If it fails, the method can return one of the error values shown in the following table.

```
Return code
```

DMUS_E_CHUNKNOTFOUND		
DMUS E INVALID BAND		
DMUS_E_INVALIDFILE		
DMUS_E_NOTADLSCOL		
E_FAIL		
E_OUTOFMEMORY		
<u>E_POINTER</u>		

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicObject8 Interface
 IDirectMusicObject8::SetDescriptor

IDirectMusicObject8::SetDescriptor

The **SetDescriptor** method 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.

Syntax

```
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 values in the following table is returned.

Return code	Description
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 error values in the following table.

Return code 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** Interface
- <u>IDirectMusicObject8::GetDescriptor</u>
- <u>IDirectMusicObject8::ParseDescriptor</u>

IDirectMusicPatternTrack8 Interface

The **IDirectMusicPatternTrack8** interface represents a track that contains a single <u>pattern</u>. A pattern track is similar to a sequence track, but because it contains <u>music values</u> rather than fixed notes, it responds to chord changes.

You can obtain this interface by passing IID_IDirectMusicPatternTrack to the **IDirectMusicTrack8::QueryInterface** method of the track.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicPatternTrack8** interface exposes the following methods.

Method	Description
CreateSegment	Creates a segment containing the pattern track.
<u>SetPatternByName</u>	Sets the pattern to be played by the track.
<u>SetVariation</u>	Sets the variations to be played by a part in the
	track.

Requirements

Header: Declared in dmusici.h.

See Also

• DirectMusic Interfaces

IDirectMusicPatternTrack8::CreateSegment

The **CreateSegment** method creates a segment containing the pattern track.

Syntax

```
HRESULT CreateSegment(
    IDirectMusicStyle* pStyle,
    IDirectMusicSegment** ppSegment
);
```

Parameters

pStyle

<u>Style</u> 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 error values shown in the following table.

Return code 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**.

This method does not assign a pattern to the track. The style in *pStyle* provides only the tempo, time signature, and band. To assign a pattern, use **IDirectMusicPatternTrack8::SetPatternByName**.

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicPatternTrack8 Interface

IDirectMusicPatternTrack8::SetPatternByName

The **SetPatternByName** method sets the pattern to be played by the track. The pattern comes from a <u>style</u>.

Syntax

```
HRESULT SetPatternByName(
    IDirectMusicSegmentState* pSegState,
    WCHAR* wszName,
    IDirectMusicStyle* pStyle,
    DWORD dwPatternType,
    DWORD* pdwLength
);
```

Parameters

pSegState

Address of the **<u>IDirectMusicSegmentState8</u>** interface representing the playing instance of the segment that contains 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 <u>style</u> containing the pattern.

dwPatternType

One of the **DMUS_STYLET_TYPES** enumerations 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 error values shown in the following table.

Return code

DMUS_E_NOT_FOUND DMUS_E_NOT_INIT E_OUTOFMEMORY E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicPatternTrack8 Interface

IDirectMusicPatternTrack8::SetVariation

The **SetVariation** method sets the variations to be played by a part in the track.

Syntax

```
HRESULT SetVariation(
    IDirectMusicSegmentState* pSegState,
    DWORD dwVariationFlags,
    DWORD dwPart
);
```

Parameters

pSegState

Address of the **<u>IDirectMusicSegmentState8</u>** interface representing the playing instance of the segment that contains 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 authoring application, and is equivalent to the <u>performance channel</u>.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the error values shown in the following table.

Return code 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.

See Also

• IDirectMusicPatternTrack8 Interface

IDirectMusicPerformance8 Interface

The **IDirectMusicPerformance8** interface is the overall manager of playback. It is used for adding and removing ports, mapping <u>performance channels</u> to ports, playing segments, dispatching messages and routing them through tools, requesting and receiving event notification, and setting and retrieving various parameters. It also has several methods for getting information about timing and for converting time and <u>music values</u> 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 have separate tempos, whereas all segments within one performance must play at the same tempo.

IDirectMusicPerformance8 supersedes the **IDirectMusicPerformance** interface and adds new methods.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicPerformance8** interface exposes the following methods, arranged by category.

Audiopaths

Method	Description
<u>CreateAudioPath</u>	Creates an audiopath from a
	configuration object.
<u>CreateStandardAudioPath</u>	Creates an audiopath with a standard
	configuration.
GetDefaultAudioPath	Retrieves the default audiopath.
SetDefaultAudioPath	Sets and activates the default audiopath
	for the performance.

Messages

Method

Description

<u>AllocPMsg</u>	Allocates memory for a performance
	message.
ClonePMsg	Makes a copy of a performance
	message.
<u>FreePMsg</u>	Frees memory allocated for a
	performance message.
SendPMsg	Sends a performance message.
	performance message.

MIDI conversion

Method	Description
MIDIToMusic	Converts a MIDI note value to a DirectMusic <u>music value</u> , using a supplied chord, subchord level, and play mode.
MusicToMIDI	Converts a DirectMusic music value to a MIDI note value.

Notifications

Method	Description
AddNotificationType	Adds a notification type to the
Autivotification Type	performance.
GetNotificationPMsg	Retrieves a pending notification
	message.
<u>RemoveNotificationType</u>	Removes a previously added
	notification type from the performance.
SetNotificationHandle	Sets the event handle for notifications.

Parameters

Method	Description
	Retrieves global values from the

<u>GetGlobalParam</u>	performance.
<u>GetParam</u>	Retrieves data from a track.
GetParamEx	Retrieves data from a track, with
	support for self-controlling segments.
SetGlobalParam	Sets global values for the performance.
SetDayam	Sets data on a track in the control
<u>SetParam</u>	segment.

Performance channels

Method	Description
AssignPChannel	Assigns a single <u>performance channel</u> to the performance and maps it to a port, group, and MIDI channel.
AssignPChannelBlock	Assigns a block of 16 performance channels to the performance and maps them to a port and a channel group.
<u>PChannelInfo</u>	Retrieves the port, group, and MIDI channel for a given performance channel.

Playback

Method	Description
<u>GetSegmentState</u>	Retrieves the currently playing primary
	segment state or the primary segment
	state that is playing at a given time.
<u>IsPlaying</u>	Ascertains whether a specified segment
	or segment state is currently being
	heard from the speakers.
PlaySegment	Begins playback of a segment.
PlaySegmentEx	Begins playback of a segment, with
	options for transition and audiopath.

<u>Stop</u>	Stops playback of a segment or segment state.
	State.
StopEx	Stops playback of a segment, segment state, or audiopath.
	;F

Ports

Method	Description
AddPort	Assigns a port to the performance.
<u>RemovePort</u>	Removes a port from the performance.

Timing

Method	Description
<u>AdjustTime</u>	Adjusts the internal performance time forward or backward.
<u>GetBumperLength</u>	Retrieves the interval 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.
<u>GetLatencyTime</u>	Retrieves the latency time, which is the performance time being heard from the speakers plus the time required to queue and render messages.
<u>GetPrepareTime</u>	Retrieves the interval between the time when messages are sent by tracks and the time when the sound is heard.
<u>GetQueueTime</u>	Retrieves the earliest time in the queue at which messages can be flushed.
<u>GetResolvedTime</u>	Resolves a given time to a given boundary.
<u>GetTime</u>	Retrieves the current time of the performance.
	Converts a performance time in

<u>MusicToReferenceTime</u>	MUSIC_TIME format to performance time in REFERENCE_TIME format.
<u>ReferenceToMusicTime</u>	Converts a performance time in REFERENCE_TIME format to a performance time in MUSIC_TIME format.
RhythmToTime	Converts rhythm time to music time.
<u>SetBumperLength</u>	Sets the interval 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.
<u>SetPrepareTime</u>	Sets the interval between the time when messages are sent by tracks and the time when the sound is heard.
TimeToRhythm	Converts music time to rhythm time.

Toolgraphs

Method	Description
<u>GetGraph</u>	Retrieves the toolgraph of a
	performance.
<u>SetGraph</u>	Replaces the performance's toolgraph.

Miscellaneous

Method	Description
<u>CloseDown</u>	Closes down the performance object.
<u>DownloadInstrument</u>	Downloads <u>DLS</u> instrument data to a
	port.
<u>InitAudio</u>	Initializes the performance and
	optionally sets up a default audiopath.
	Flushes all queued messages from the
	supplied time forward and causes all
<u>Invalidate</u>	tracks of all segments to resend their

Requirements

Header: Declared in dmusici.h.

See Also

• <u>DirectMusic Interfaces</u>

IDirectMusicPerformance8::AddNotificationType

The **AddNotificationType** method adds a notification type to the performance. All segments and tracks are automatically updated with the new notification by an internal call to their **AddNotificationType** methods.

Syntax

```
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 error values shown in the following table.

Return code <u>E_OUTOFMEMORY</u> <u>E_POINTER</u>

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
- <u>IDirectMusicPerformance8::RemoveNotificationType</u>
- <u>IDirectMusicSegment8::AddNotificationType</u>
- IDirectMusicTrack8::AddNotificationType
- Notification and Event Handling

IDirectMusicPerformance8::AddPort

The **AddPort** method assigns a port to the performance. This method is valid only for performances that do not use audiopaths. Such performances are initialized by using **IDirectMusicPerformance8::Init**.

Syntax

```
HRESULT AddPort(
    IDirectMusicPort* pPort
);
```

Parameters

pPort

IDirectMusicPort8 interface pointer of 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 error values shown in the following table.

Return codeDMUS_E_CANNOT_OPEN_PORTDMUS_E_NOT_INITE_OUTOFMEMORYE_POINTER

Remarks

When the default port is specified by passing NULL in *pPort*, it is assigned one channel group. If no <u>performance channels</u> 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** Interface
- IDirectMusicPerformance8::RemovePort
- <u>Default Port</u>

IDirectMusicPerformance8::AdjustTime

The **AdjustTime** method adjusts the internal performance time forward or backward. This is mostly used to compensate for drift when synchronizing to another source.

Syntax

```
HRESULT AdjustTime(
    REFERENCE_TIME rtAmount
);
```

Parameters

rtAmount

Amount of time to add. This can be a value in the range from -10,000,000 to 10,000,000 (-1 second to +1 second).

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return **<u>E_INVALIDARG</u>**.

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 Interface

- IDirectMusicPerformance8::GetTime
 Timing

IDirectMusicPerformance8::AllocPMsg

The **AllocPMsg** method allocates memory for a performance message.

Syntax

```
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 error values shown in the following table.

Return code 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** Interface
- IDirectMusicPerformance8::FreePMsg
- <u>IDirectMusicPerformance8::SendPMsg</u>
- <u>DirectMusic Messages</u>

IDirectMusicPerformance8::AssignPChannel

The **AssignPChannel** method assigns a single <u>performance channel</u> 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.

Syntax

```
HRESULT AssignPChannel(
   DWORD dwPChannel,
   IDirectMusicPort* pPort,
   DWORD dwGroup,
   DWORD dwMChannel
);
```

Parameters

dwPChannel

Performance channel to assign.

pPort

Pointer to the **IDirectMusicPort8** interface of 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 <u>S_FALSE</u> (see Remarks).

If it fails, the method can return one of the error values shown in the following table.

Return code

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

- <u>Channels</u>
- **<u>IDirectMusicPerformance8</u>** Interface
- IDirectMusicPerformance8::AssignPChannelBlock
- IDirectMusicPerformance8::PChannelInfo

IDirectMusicPerformance8::AssignPChannelBlock

The **AssignPChannelBlock** method assigns a block of 16 <u>performance channels</u> 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 <u>IDirectMusicPerformance8::Init</u>.

Syntax

```
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

IDirectMusicPort8 interface pointer of 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 error values shown in the following table.

Return code 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 <u>S_FALSE</u> 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

- <u>Channels</u>
- <u>Default Port</u>
- **IDirectMusicPerformance8** Interface
- IDirectMusicPerformance8::AssignPChannel
- IDirectMusicPerformance8::PChannelInfo

IDirectMusicPerformance8::ClonePMsg

The **ClonePMsg** method makes a copy of a performance message.

Syntax

```
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 error values shown in the following table.

Return code <u>E_OUTOFMEMORY</u> <u>E_POINTER</u>

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicPerformance8 Interface

IDirectMusicPerformance8::CloseDown

The **CloseDown** method 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.

Syntax

HRESULT CloseDown();

Parameters

This method returns no parameters.

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 device object was created in the call to **Init** or **InitAudio** but no reference was returned to the application, **CloseDown** also releases the DirectSound device 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 the DirectSound device object explicitly, or obtained a reference form **Init** or **InitAudio**, it is responsible for releasing the DirectSound device.

The method releases any downloaded instruments that have not been unloaded.

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicPerformance8 Interface

IDirectMusicPerformance8::CreateAudioPath

The **CreateAudioPath** method creates an audiopath from a configuration object.

Syntax

```
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 **<u>IDirectMusicAudioPath8</u>** 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 error values shown in the following table.

 Return code

 DMUS_E_AUDIOPATHS_NOT_VALID

DMUS_E_NOT_INIT
DSERR_BUFFERLOST
E_NOINTERFACE
E_OUTOFMEMORY
<u>E_POINTER</u>

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

- IDirectMusicPerformance8 Interface
- IDirectMusicPerformance8::CreateStandardAudioPath
- <u>Creating Audiopaths</u>

IDirectMusicPerformance8::CreateStandardAudioPath

The **CreateStandardAudioPath** method creates an audiopath with a standard configuration.

Syntax

```
HRESULT CreateStandardAudioPath(
   DWORD dwType,
   DWORD dwPChannelCount,
   BOOL fActivate,
   IDirectMusicAudioPath **ppNewPath
);
```

Parameters

dwType

Type of the path. The following values are defined.

Value	Description
	One bus to a 3-D buffer.
DMUS_APATH_DYNAMIC_3D	Does not send to
	environmental reverb.
DMUS_APATH_DYNAMIC_MONO	One bus to a mono buffer.
DMUS_APATH_DYNAMIC_STEREO	Two buses to a stereo
	buffer.
	Ordinary music setup
DMUS_APATH_SHARED_STEREOPLUSREVERE	with stereo outs and
	reverb.

dwPChannelCount

Number of <u>performance channels</u> 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 error values shown in the following table.

Return code

DMUS_E_AUDIOPATHS_NOT_VALID
DMUS_E_NOT_INIT
DSERR_BUFFERLOST
E_INVALIDARG
E_OUTOFMEMORY
E_POINTER

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.

See Also

- **IDirectMusicPerformance8** Interface
- <u>IDirectMusicPerformance8::CreateAudioPath</u>
- **Standard Audiopaths**

IDirectMusicPerformance8::DownloadInstrument

The **DownloadInstrument** method downloads <u>DLS</u> instrument data to a port.

Syntax

```
HRESULT DownloadInstrument(
    IDirectMusicInstrument* pInst,
    DWORD dwPChannel,
    IDirectMusicDownloadedInstrument** ppDownInst,
    DMUS_NOTERANGE* pNoteRanges,
    DWORD dwNumNoteRanges,
    IDirectMusicPort** ppPort,
    DWORD* pdwGroup,
    DWORD* pdwGroup,
    DWORD* pdwMChannel
);
```

Parameters

pInst

Pointer to the **IDirectMusicInstrument8** interface of the instrument to download.

dwPChannel

<u>Performance channel</u> 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 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 channel group to which the instrument was assigned.

pdwMChannel

Address of a variable that receives the MIDI channel to which the instrument was assigned.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the error values shown in the following table.

Return code

E_INVALIDARG E_POINTER

Remarks

Most applications do not need to use this method, because instrument downloading is normally handled by 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

- <u>Downloading and Unloading Bands</u>
- **IDirectMusicPerformance8** Interface
- <u>IDirectMusicPort8::DownloadInstrument</u>
- IDirectMusicPort8::UnloadInstrument
- Working with Instruments

IDirectMusicPerformance8::FreePMsg

The **FreePMsg** method frees memory allocated for a performance message.

Syntax

```
HRESULT FreePMsg(
    DMUS_PMSG* pPMSG
);
```

Parameters

pPMSG

Pointer to the 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 error values shown in the following table.

Return code <u>DMUS_E_CANNOT_FREE</u> <u>E_POINTER</u>

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
 <u>IDirectMusicPerformance8::SendPMsg</u> 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

- <u>DirectMusic Messages</u>
- **IDirectMusicPerformance8** Interface
- IDirectMusicPerformance8::AllocPMsg

IDirectMusicPerformance8::GetBumperLength

The **GetBumperLength** method retrieves the interval 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.

Syntax

```
HRESULT GetBumperLength(
    DWORD* pdwMilliSeconds
);
```

Parameters

pdwMilliSeconds

Address of a variable to receive the buffer length.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return <u>E_POINTER</u>.

Remarks

The default value is 50 milliseconds.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
- <u>IDirectMusicPerformance8::SetBumperLength</u>
- Latency and Bumper Time

IDirectMusicPerformance8::GetDefaultAudioPath

The **GetDefaultAudioPath** method retrieves the default audiopath set by **IDirectMusicPerformance8::InitAudio** or **IDirectMusicPerformance8::SetDefaultAudioPath**.

Syntax

```
HRESULT GetDefaultAudioPath(
    IDirectMusicAudioPath** ppAudioPath
);
```

Parameters

ppAudioPath

Address of a variable that receives the **<u>IDirectMusicAudioPath8</u>** 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 error values shown in the following table.

Return code DMUS_E_AUDIOPATHS_NOT_VALID DMUS_E_NOT_INIT E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

- Default Audiopath
 IDirectMusicPerformance8 Interface

IDirectMusicPerformance8::GetGlobalParam

The **GetGlobalParam** method retrieves a global parameter that has been set by **IDirectMusicPerformance8::SetGlobalParam**.

Syntax

```
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 error values shown in the following table.

Return code

E_INVALIDARG E_POINTER

Remarks

Do not assume that any parameter has a default value that can be retrieved by using **GetGlobalParam**. 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 **GetGlobalParam** might return E_INVALIDARG.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
- IDirectMusicPerformance8::GetParam
- <u>IDirectMusicPerformance8::SetGlobalParam</u>
- <u>Performance Parameters</u>

IDirectMusicPerformance8::GetGraph

The **GetGraph** method retrieves the toolgraph of a performance.

Syntax

```
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 error values shown in the following table.

Return code DMUS_E_NOT_FOUND E_POINTER

Remarks

The reference count of the graph is incremented.

The performance object implements **IDirectMusicGraph** directly This interface is used primarily to call the **IDirectMusicGraph8::StampPMsg** method directly on the performance. This has nothing to do with a graph object that might be embedded in the performance. If you want to access an embedded object, you are accessing a different **IDirectMusicGraph** interface because it is an interface on the embedded graph object, not the performance itself. To obtain an interface to an embedded audiopath, use the **IDirectMusicAudioPath8::GetObjectInPath** or **IDirectMusicSegmentState8::GetObjectInPath** method.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Tools</u>
- **IDirectMusicPerformance8** Interface
- <u>IDirectMusicPerformance8::SendPMsg</u>
- IDirectMusicPerformance8::SetGraph
- IDirectMusicSegment8::GetGraph

IDirectMusicPerformance8::GetLatencyTime

The **GetLatencyTime** method retrieves the latency time, which is the performance time being heard from the speakers plus the time required to queue and render messages.

Syntax

```
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 error values shown in the following table.

 Return code

 DMUS_E_NO_MASTER_CLOCK

 E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
- Latency and Bumper Time

IDirectMusicPerformance8::GetNotificationPMsg

The **GetNotificationPMsg** method retrieves a pending notification message.

Syntax

```
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 <u>S_FALSE</u> if there are no more notification events to return.

If it fails, the method can return **<u>E_POINTER</u>**.

Remarks

For most notifications, the segment state that generated the notification can be retrieved from the **punkUser** member of the message structure, and the segment can be retrieved by using **IDirectMusicSegmentState8::GetSegment**.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
 Notification and Event Handling

IDirectMusicPerformance8::GetParam

The **GetParam** method retrieves data from a track inside the control segment.

Syntax

```
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 retrieve. See <u>Standard Track Parameters</u>.

dwGroupBits

Group of the track from which to retrieve the data (see Remarks). Set this value to 0xFFFFFFF for all groups.

dwIndex

Index of the track in the group from which to retrieve the data, or DMUS_SEG_ANYTRACK to find the first track that contains the parameter.

mtTime

Time from which to retrieve 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 error values shown in the following table.

Return code

DMUS	<u>E_GET_UNSUPPORTED</u>
DMUS	E_NO_MASTER_CLOCK
DMUS	<u>E_NOT_FOUND</u>
DMUS	E TRACK NOT FOUND
E_POIN	<u>ITER</u>

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 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 <u>Identifying the Track</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>Control Segments</u>
- IDirectMusicPerformance8 Interface
- <u>IDirectMusicPerformance8::GetParamEx</u>
- <u>IDirectMusicPerformance8::GetTime</u>
- <u>IDirectMusicPerformance8::SetGlobalParam</u>
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- IDirectMusicTrack8::GetParamEx
- <u>Performance Parameters</u>

IDirectMusicPerformance8::GetParamEx

The **GetParamEx** method 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.

Syntax

```
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 <u>Standard Track Parameters</u>.

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 from one 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 error values shown in the following table.

Return codeDMUS_E_GET_UNSUPPORTEDDMUS_E_NO_MASTER_CLOCKDMUS_E_NOT_FOUNDDMUS_E_TRACK_NOT_FOUNDE_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 <u>Identifying the Track</u>.

The data returned in **pParam* can become invalid before the time returned in **pmtNext* if another control segment is cued.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>Control Segments</u>
- IDirectMusicPerformance8 Interface
- <u>IDirectMusicPerformance8::GetTime</u>
- <u>IDirectMusicPerformance8::SetGlobalParam</u>
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicTrack8::GetParamEx</u>
- <u>Performance Parameters</u>
- <u>Self-Controlling Segments</u>

IDirectMusicPerformance8::GetPrepareTime

The **GetPrepareTime** method 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.

Syntax

```
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 <u>E_POINTER</u>.

Remarks

The default value is 1000 milliseconds.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
- <u>IDirectMusicPerformance8::SetPrepareTime</u>
- <u>Prepare Time</u>

IDirectMusicPerformance8::GetQueueTime

The **GetQueueTime** method 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.

Syntax

```
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 error values shown in the following table.

Return code DMUS_E_NO_MASTER_CLOCK E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

• **IDirectMusicPerformance8** Interface

- IDirectMusicPerformance8::Invalidate
 Latency and Bumper Time

IDirectMusicPerformance8::GetResolvedTime

The **GetResolvedTime** method resolves a given time to a given boundary.

Syntax

```
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 **<u>E_POINTER</u>**.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
 Segment Timing

IDirectMusicPerformance8::GetSegmentState

The **GetSegmentState** method retrieves the currently playing primary segment state or the primary segment state that is playing at a given time.

Syntax

```
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 error values shown in the following table.

Return codeDMUS_E_NOT_FOUNDE_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.

See Also

- IDirectMusicPerformance8 Interface
- <u>Segment States</u>

IDirectMusicPerformance8::GetTime

The **GetTime** method retrieves the current time of the performance. Events cued at this time are now being performed.

Syntax

```
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 error values shown in the following table.

Return code DMUS_E_NO_MASTER_CLOCK E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
- <u>Timing</u>

IDirectMusicPerformance8::Init

The **Init** method associates the performance with a DirectMusic object and a DirectSound device 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**.

Syntax

```
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 waveform output. If this value is NULL, DirectMusic creates a DirectSound device object. There should, however, be only one DirectSound device 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 error values shown in the following table.

Return code

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

• **IDirectMusicPerformance8** Interface

IDirectMusicPerformance8::InitAudio

The **InitAudio** method 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**.

Syntax

```
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 device object to use by default for waveform output. If this parameter contains a NULL pointer, DirectMusic creates a private DirectSound device object. If the variable pointed to contains NULL, DirectMusic creates a DirectSound device 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

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 <u>performance channels</u> 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 values listed in the following table are defined for use in this parameter.

Value	Description
	3-D buffers. This flag is not

DMUS_AUDIOF_3D	implemented. Buffers in 3-D audiopaths always have 3-D capabilities.
DMUS_AUDIOF_ALL	All features.
DMUS_AUDIOF_BUFFERS	Multiple buffers.
DMUS_AUDIOF_DMOS	Additional DMOs. This flag is not implemented.
DMUS_AUDIOF_ENVIRON	Environmental modeling. This flag is not implemented.
DMUS_AUDIOF_EAX	Support for Environmental Audio Extensions (EAX). This flag is not implemented.
DMUS_AUDIOF_STREAMING	Support for streaming waveforms.

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 error values shown in the following table.

Return code

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. The DirectSound device 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 **<u>IDirectMusicPerformance8::CloseDown</u>** method before being released.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>Creating the Performance</u>
 <u>IDirectMusicPerformance8</u> Interface

IDirectMusicPerformance8::Invalidate

The **Invalidate** method flushes all queued messages from the specified time forward and causes all tracks of all segments to resend their data.

Syntax

```
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

Flag that aligns *mtTime* to the next measure, beat, or grid. 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 <u>DMUS_E_NO_MASTER_CLOCK</u>.

Remarks

If *mtTime* is so long ago that it is impossible to invalidate that time, the earliest possible time is used.

Notes that have already been sent to the port are normally cut off by invalidation; that is, any pending note-off message is immediately sent. However, this behavior can be overridden by using one of the **DMUS_NOTEF_FLAGS** flags in the message structure.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicPerformance8** Interface
- Latency and Bumper Time
- <u>Prepare Time</u>
- <u>Segment Timing</u>

IDirectMusicPerformance8::IsPlaying

The **IsPlaying** method ascertains whether a specified segment or segment state is currently being heard from the speakers.

Syntax

```
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 error values shown in the following table.

Return codeE_POINTERDMUS_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.

See Also

• IDirectMusicPerformance8 Interface

IDirectMusicPerformance8::MIDIToMusic

The **MIDIToMusic** method converts a MIDI note value to a DirectMusic <u>music value</u>, using a supplied chord, subchord level, and play mode.

Syntax

```
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.

Return code

S_OK DMUS_S_DOWN_OCTAVE DMUS_S_UP_OCTAVE

If it fails, the method can return one of the error values shown in the following table.

Return code 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 adjusted 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** Interface
- IDirectMusicPerformance8::MusicToMIDI
- Music Values and MIDI Notes

IDirectMusicPerformance8::MusicToMIDI

The **MusicToMIDI** method converts a DirectMusic <u>music value</u> to a MIDI note value.

Syntax

```
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.

Return code

S_OK DMUS_S_OVER_CHORD DMUS_S_DOWN_OCTAVE DMUS_S_UP_OCTAVE

If it fails, the method can return one of the error values shown in the following table.

Return code

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 adjusted 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** Interface
- IDirectMusicPerformance8::MIDIToMusic
- Music Values and MIDI Notes

IDirectMusicPerformance8::MusicToReferenceTime

The **MusicToReferenceTime** method converts a performance time in **MUSIC_TIME** format to performance time in **REFERENCE_TIME** format.

Syntax

```
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 error values shown in the following table.

Return codeE_POINTERDMUS_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

- <u>Clock Time and Music Time</u>
- IDirectMusicPerformance8 Interface
- <u>IDirectMusicPerformance8::ReferenceToMusicTime</u>

IDirectMusicPerformance8::PChannelInfo

The **PChannelInfo** method retrieves the port, group, and MIDI channel for a given <u>performance channel</u>.

Syntax

```
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 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 error values shown in the following table.

Return code

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** Interface
- <u>IDirectMusicPerformance8::AssignPChannel</u>
- IDirectMusicPerformance8::AssignPChannelBlock

IDirectMusicPerformance8::PlaySegment

The **PlaySegment** method begins playback of a segment.

Syntax

```
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 error values shown in the following table.

Return code

<u>E_OUTOFMEMORY</u>		
E_POINTER		
DMUS_E_NO_MASTER_CLOCK		
DMUS E SEGMENT INIT FAILED		
DMUS E TIME PAST		

Remarks

Do not play segments from untrusted sources. Improperly written segments can make excessive demands on system resources, resulting in degradation of performance or system failure.

Segments should be greater than 250 milliseconds in length.

The boundary resolutions in *dwFlags* are relative to the current 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 <u>Segment Timing</u>. For information on how the start time of segments can be affected by tempo changes, see <u>Clock Time and Music Time</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicPerformance8** Interface
- <u>IDirectMusicPerformance8::PlaySegmentEx</u>
- <u>Playing Segments</u>

IDirectMusicPerformance8::PlaySegmentEx

The **PlaySegmentEx** method begins playback of a segment. The method offers greater functionality than **IDirectMusicPerformance8::PlaySegment**.

Syntax

```
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

Reserved. Set to NULL.

pTransition

IUnknown interface pointer of a <u>template</u> 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. See Remarks.

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 error values shown in the following table.

Return code

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

Do not play segments from untrusted sources. Improperly written segments can make excessive demands on system resources, resulting in degradation of performance or system failure.

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 <u>Segment Timing</u>. For information on how the start time of segments can be affected by tempo changes, see <u>Clock Time and Music Time</u>.

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 on 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.

When the segment is being cued as a secondary segment, the *pFrom* parameter can be used to specify the segment state of another secondary segment against which to cue the new segment, as in the following example function.

```
HRESULT CueOneAfterAnother(IDirectMusicSegment8* pSegmentA,
            IDirectMusicSegmentState8* pSegmentStateB,
            IDirectMusicPerformance8* pPerf)
{
        HRESULT hr = pPerf->PlaySegmentEx(
            pSegmentA,
            NULL, NULL,
            DMUS_SEGF_QUEUE | DMUS_SEGF_SECONDARY,
            0, NULL,
            pSegmentStateB,
            NULL);
```

```
return hr;
}
```

In the example, *pSegmentStateB* plays to the end before stopping because the DMUS_SEGF_QUEUE flag has been set. If you set a different flag such as DMUS_SEGF_MEASURE, *pSegmentStateB* stops as soon as the boundary is reached and the new secondary segment begins playing.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
- <u>Playing Segments</u>

IDirectMusicPerformance8::ReferenceToMusicTime

The **ReferenceToMusicTime** method converts a performance time in **REFERENCE_TIME** format to a performance time in **MUSIC_TIME** format.

Syntax

```
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 error values shown in the following table.

Return codeE_POINTERDMUS_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

- <u>Clock Time and Music Time</u>
- **IDirectMusicPerformance8** Interface
- <u>IDirectMusicPerformance8::MusicToReferenceTime</u>

IDirectMusicPerformance8::RemoveNotificationType

The **RemoveNotificationType** method removes a previously added notification type from the performance. All segments and tracks are updated by a call to their **RemoveNotificationType** methods.

Syntax

```
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 removed.

Return Values

If the method succeeds, the return value is S_OK or <u>S_FALSE</u> (see Remarks).

If it fails, the method can return **<u>E_POINTER</u>**.

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** Interface
- <u>IDirectMusicPerformance8::AddNotificationType</u>
- <u>IDirectMusicSegment8::RemoveNotificationType</u>
- IDirectMusicTrack8::RemoveNotificationType
- Notification and Event Handling

IDirectMusicPerformance8::RemovePort

The **RemovePort** method removes a port from the performance. Any <u>performance channels</u> that map to this port are invalidated, and messages on those channels are not performed.

Syntax

```
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 error values shown in the following table.

Return code

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 Interface
- <u>IDirectMusicPerformance8::AddPort</u>

IDirectMusicPerformance8::RhythmToTime

The **RhythmToTime** method converts rhythm time to music time.

Syntax

```
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 **<u>E_POINTER</u>**.

Remarks

The method calculates a duration from the supplied measure, beat, grid, and offset, and adds the value in the **mtTime** member of the **DMUS_TIMESIGNATURE** structure.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicPerformance8** Interface
- <u>IDirectMusicPerformance8::TimeToRhythm</u>

IDirectMusicPerformance8::SendPMsg

The **SendPMsg** method sends a performance message. This method is called by tracks when they are played. It can also be called by an application or tool to insert data into a performance.

Syntax

```
HRESULT SendPMsg(
    DMUS_PMSG* pPMSG
);
```

Parameters

pPMSG

Message allocated by **IDirectMusicPerformance8::AllocPMsg**. This structure is of a type derived from **DMUS_PMSG**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the error values shown in the following table.

Return code

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

whether the time stamp is in **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. If the message is not stamped, it is not delivered to any tools implemented by the application, and might not play on the correct performance channel. If you want the message to pass only through the performance toolgraph, obtain the **IDirectMusicGraph8** interface by calling **IDirectMusicPerformance8::QueryInterface**. Otherwise, obtain it by calling **IDirectMusicSegmentState8::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.

If you are using audiopaths, it is generally best not to send a message directly to the performance, because the performance channels may have been remapped, and messages sent to a particular channel may not be played. Instead, obtain the **IDirectMusicGraph8** interface from **IDirectMusicAudioPath8**. The audiopath then manages the remapping of the performance channel.

Normally, the performance frees the message after it has been processed. For more information, see the Remarks for **IDirectMusicPerformance8::FreePMsg**.

The following example function shows how to allocate and send a system exclusive message to the performance graph.

```
if (SUCCEEDED(hr = pPerformance->QueryInterface(IID_IDirectMusicGr
                                                    (void**)&pGraph)))
  {
    DMUS_SYSEX_PMSG* pSysEx;
    if (SUCCEEDED(hr = pPerformance->AllocPMsg(
                    sizeof(DMUS_SYSEX_PMSG) + dwSysExLength,
                    (DMUS_PMSG**)&pSysEx)))
    {
      ZeroMemory(pSysEx, sizeof(DMUS_SYSEX_PMSG));
      pSysEx -> dwLen = dwSysExLength;
      pSysEx->mtTime = mtTime;
      pSysEx->dwFlags = DMUS_PMSGF_MUSICTIME;
      pSysEx->dwType = DMUS_PMSGT_SYSEX;
      memcpy(pSysEx->abData, pbSysExData, dwSysExLength);
      pGraph->StampPMsg((DMUS_PMSG*)pSysEx);
      if (FAILED(hr = pPerformance->SendPMsg((DMUS_PMSG*)pSysEx)))
      {
        pPerformance->FreePMsg( (DMUS_PMSG*)pSysEx );
      }
    }
    pGraph->Release();
  }
  return hr;
}
```

The next example function 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->rtTime = 0;
                                          // Ignored.
                                          // When to play the note.
  pNote->mtTime = mtTime;
 pNote->dwFlags = DMUS_PMSGF_MUSICTIME; // Use the mtTime field.
                                          // Play on PChannel 5.
  pNote -> dwPChannel = 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 = 0xFFFFFFF; // 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, 0xFFFFFFF,
  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,
    Θ,
    &(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.
                                  // MIDI velocity (0 to 127).
pNote->bVelocity = 120;
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.
                                   // Don't randomize velocity.
pNote - bVelRange = 0;
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

}

- DirectMusic Messages
- <u>DirectMusic Tools</u>
- IDirectMusicPerformance8 Interface
- <u>IDirectMusicTool8::ProcessPMsg</u>
- <u>Messages</u>

IDirectMusicPerformance8::SetBumperLength

The **SetBumperLength** method sets the interval 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.

Syntax

```
HRESULT SetBumperLength(
   DWORD dwMilliSeconds
);
```

Parameters

dwMilliSeconds

Buffer length, in milliseconds. The default value is 50.

Return Values

The method returns S_OK.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
- IDirectMusicPerformance8::GetBumperLength
- <u>IDirectMusicPerformance8::SetPrepareTime</u>
- Latency and Bumper Time

IDirectMusicPerformance8::SetDefaultAudioPath

The **SetDefaultAudioPath** method sets and activates the default audiopath for the performance.

Syntax

```
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 error values shown in the following table.

Return code

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

- Default Audiopath
- **IDirectMusicPerformance8** Interface
- IDirectMusicPerformance8::GetDefaultAudioPath
- IDirectMusicPerformance8::InitAudio

IDirectMusicPerformance8::SetGlobalParam

The **SetGlobalParam** method sets global values for the performance.

Syntax

```
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 error values shown in the following table.

Return code

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 <u>Setting and Retrieving Global Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicPerformance8 Interface
- IDirectMusicPerformance8::GetGlobalParam
- IDirectMusicPerformance8::SetParam
- <u>Performance Parameters</u>

IDirectMusicPerformance8::SetGraph

The **SetGraph** method replaces the performance's toolgraph.

Syntax

```
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 <u>E_POINTER</u>.

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

- <u>DirectMusic Tools</u>
- **IDirectMusicPerformance8** Interface
- <u>IDirectMusicPerformance8::GetGraph</u>

- IDirectMusicPerformance8::SendPMsg
 IDirectMusicSegment8::SetGraph

IDirectMusicPerformance8::SetNotificationHandle

The **SetNotificationHandle** method sets the event handle for notifications.

Syntax

```
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

When the event is signaled, the application should call the **IDirectMusicPerformance8::GetNotificationPMsg** method to retrieve the notification event.

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

- IDirectMusicPerformance8 Interface
- Notification and Event Handling

IDirectMusicPerformance8::SetParam

The **SetParam** method sets data on a track in the control segment.

Syntax

```
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 <u>Standard Track Parameters</u>.

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 error values shown in the following table.

Return code

DMUS	E_NO_MASTER_CLOCK		
DMUS	<u>E_SET_UNSUPPORTED</u>		
<u>DMUS</u>	<u>E_TRACK_NOT_FOUND</u>		
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.

For an explanation of *dwGroupBits* and *dwIndex*, see <u>Identifying the Track</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>Control Segments</u>
- IDirectMusicPerformance8 Interface
- IDirectMusicPerformance8::GetParam
- IDirectMusicPerformance8::GetTime
- <u>IDirectMusicPerformance8::SetGlobalParam</u>
- IDirectMusicSegment8::SetParam
- IDirectMusicTrack8::SetParamEx
- <u>Performance Parameters</u>

IDirectMusicPerformance8::SetPrepareTime

The **SetPrepareTime** method 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.

Syntax

```
HRESULT SetPrepareTime(
   DWORD dwMilliSeconds
);
```

Parameters

dwMilliSeconds

Amount of prepare time, in milliseconds. The default value is 1000.

Return Values

The method returns S_OK.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicPerformance8** Interface
- <u>IDirectMusicPerformance8::GetPrepareTime</u>
- IDirectMusicPerformance8::SetBumperLength
- <u>Prepare Time</u>

IDirectMusicPerformance8::Stop

The **Stop** method 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.

Syntax

```
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 **<u>E_POINTER</u>**.

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 <u>performance channels</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DMUS_SEGF_FLAGS</u>
- **<u>IDirectMusicPerformance8</u>** Interface
- <u>IDirectMusicPerformance8::PlaySegmentEx</u>
- <u>IDirectMusicPerformance8::PlaySegment</u>
- <u>IDirectMusicPerformance8::StopEx</u>

IDirectMusicPerformance8::StopEx

The **StopEx** method stops playback of a segment, segment state, or audiopath.

Syntax

```
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 adjust the time when the stop should occur. Boundaries are in relation to the current primary segment. Can be 0 or one of the following values. DMUS_SEGF_REFTIME can be combined with one other flag.

Value	Description
DMUS_SEGF_AUTOTRANSITION	Not implemented.
DMUS_SEGF_BEAT	Stop on the next beat boundary at or after <i>i64StopTime</i> .
DMUS_SEGF_DEFAULT	Stop on the default boundary, as set by the IDirectMusicSegment8::SetDefaultResol method.

DMUS_SEGF_GRID	Stop on the next grid boundary at or after <i>i64StopTime</i> .
DMUS_SEGF_MEASURE	Stop on the next measure boundary at or af <i>i64StopTime</i> .
DMUS_SEGF_REFTIME	The value in <i>i64StopTime</i> is in reference tir
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 <u>E_POINTER</u>.

Remarks

Stopping a segment stops all instances that are playing. Stopping an audiopath stops all segments playing on that audiopath.

Requirements

Header: Declared in dmusici.h.

See Also

- DMUS_SEGF_FLAGS
- IDirectMusicPerformance8 Interface
- IDirectMusicPerformance8::PlaySegmentEx
- <u>IDirectMusicPerformance8::Stop</u>

IDirectMusicPerformance8::TimeToRhythm

The **TimeToRhythm** method converts music time to rhythm time.

Syntax

```
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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicPerformance8** Interface
- <u>IDirectMusicPerformance8::RhythmToTime</u>

IDirectMusicPort8 Interface

The **IDirectMusicPort8** interface represents a device that sends or receives sound data. Examples are the input port of an MPU-401, the output port of an MPU-401, the Microsoft software synthesizer, and an IHV-provided filter. 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 **<u>IDirectMusic8::CreatePort</u>** method.

For an overview, see <u>Using DirectMusic Ports</u>.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicPort8** interface exposes the following methods, arranged by category.

Buffers

Method	Description
<u>PlayBuffer</u>	Cues a buffer for playback by the port.
Read	Fills a buffer with incoming MIDI data.
	Specifies an event that is to be set when
SetReadNotificationHandle	MIDI messages are available to be read
	with the Read method.

Devices

Method	Description
Activate	Activates or deactivates the port.
	Calls the Win32 DeviceIoControl
DeviceIoControl	function on the underlying file handle
	implementing the port.

<u>SetDirectSound</u>	Overrides the default DirectSound device object or buffer, or both, to which a port's waveform data is
	streamed.

DLS data

Method	Description
<u>Compact</u>	Instructs the port to compact <u>DLS</u> or wave-table memory, thus making the largest possible contiguous chunk of memory available for new instruments to be downloaded.
DownloadInstrument	Downloads an instrument to the DLS device.
<u>UnloadInstrument</u>	Unloads a previously downloaded DLS instrument.

Information retrieval

Description
Retrieves the port's capabilities.
Retrieves information about the WAV
format specified in the
DMUS_PORTPARAMS8 structure
passed to IDirectMusic8::CreatePort ,
and the recommended size of the buffer
to use for waveform output.
Retrieves a pointer to the port's latency
clock.
Retrieves information about the state of
the synthesizer.

MIDI Channels

Method	Description
<u>GetChannelPriority</u>	Retrieves the priority of a MIDI
	channel.
<u>GetNumChannelGroups</u>	Retrieves the number of channel groups
	on the port.
SetChannelPriority	Sets the priority of a MIDI channel.
SetNumChannelGroups	Changes the number of channel groups
	that the application needs on the port.

The **LPDIRECTMUSICPORT8** type is defined as a pointer to the **IDirectMusicPort8** interface.

typedef IDirectMusicPort8 *LPDIRECTMUSICPORT8;

Requirements

Header: Declared in dmusicc.h.

See Also

• <u>DirectMusic Interfaces</u>

IDirectMusicPort8::Activate

The **Activate** method activates or deactivates the port.

Syntax

```
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

- **IDirectMusicPort8** Interface
- <u>IDirectMusic8::Activate</u>
- <u>Using DirectMusic Ports</u>

IDirectMusicPort8::Compact

The **Compact** method instructs the port to compact <u>DLS</u> or wave-table memory, thus making the largest possible contiguous chunk of memory available for new instruments to be downloaded.

Syntax

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 error values shown in the following table.

Return codeE_FAILE_INVALIDARGE_NOTIMPLE_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 <u>E_NOTIMPL</u>.

Requirements

Header: Declared in dmusicc.h.

See Also

• **IDirectMusicPort8** Interface

IDirectMusicPort8::DeviceIoControl

The **DeviceIoControl** method calls the Win32 **DeviceIoControl** function on the underlying file handle implementing the port.

Syntax

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. **OVERLAPPED** is defined in Winbase.h.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the error values shown in the following table.

Return code

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.

See Also

• **IDirectMusicPort8** Interface

IDirectMusicPort8::DownloadInstrument

The **DownloadInstrument** method downloads an instrument to the <u>DLS</u> device. Downloading an instrument means handing the data that makes up the instrument to the DLS device. This includes <u>articulation</u> data and all waveforms needed by the instrument. To conserve resources, only waveforms and articulation required for a range are downloaded. The method returns an **IDirectMusicDownloadedInstrument8** interface pointer, which is later used to unload the instrument.

Syntax

```
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 waveform 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 error values shown in the following table.

Return code

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

- **<u>IDirectMusicPort8</u>** Interface
- <u>IDirectMusicPort8::Compact</u>
- <u>Working with Instruments</u>

IDirectMusicPort8::GetCaps

The **GetCaps** method retrieves the port's capabilities.

Syntax

```
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 error values shown in the following table.

Return code

E_POINTER E_INVALIDARG

Requirements

Header: Declared in dmusicc.h.

See Also

• IDirectMusicPort8 Interface

IDirectMusicPort8::GetChannelPriority

The **GetChannelPriority** method retrieves the priority of a MIDI channel. For an overview, see <u>Channels</u>.

Syntax

```
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.

Value
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 <u>DLS</u> standard. They are listed here in descending order. Channel 10, the percussion channel, has the highest priority.

Value

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_HIGH_VOICE_PRIORITY combined with DAUD_CHAN10_VOICE_PRIORITY_OFFSET represents the highest priority within the high range. Combined range and offset values for the standard range are defined for convenience in Dmusicc.h as DAUD_CHAN1_DEF_VOICE_PRIORITY, .DAUD_CHAN2_DEF_VOICE_PRIORITY, and so on.

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 Interface
- IDirectMusicPort8::SetChannelPriority

IDirectMusicPort8::GetFormat

The **GetFormat** method retrieves information about the WAV format specified in the <u>DMUS_PORTPARAMS8</u> structure passed to <u>IDirectMusic8::CreatePort</u>, and the recommended size of the buffer to use for waveform output. The information can be used to create a compatible DirectSound buffer for the port.

Syntax

```
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 **<u>E_POINTER</u>**.

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** Interface
- <u>IDirectMusicPort8::SetDirectSound</u>

IDirectMusicPort8::GetLatencyClock

The **GetLatencyClock** method 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.

Syntax

```
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 <u>E_POINTER</u>.

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

- **<u>IDirectMusicPort8</u>** Interface
- Latency and Bumper Time

IDirectMusicPort8::GetNumChannelGroups

The **GetNumChannelGroups** method retrieves the number of channel groups on the port.

Syntax

```
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 error values shown in the following table.

Return code

E_FAIL E_INVALIDARG E_NOTIMPL E_OUTOFMEMORY

Requirements

Header: Declared in dmusicc.h.

See Also

• <u>Channels</u>

- IDirectMusicPort8 Interface
 IDirectMusicPort8::SetNumChannelGroups

IDirectMusicPort8::GetRunningStats

The **GetRunningStats** method retrieves information about the state of the synthesizer.

Syntax

```
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 error values shown in the following table.

Return code

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.

See Also

• **IDirectMusicPort8** Interface

IDirectMusicPort8::PlayBuffer

The **PlayBuffer** method cues a buffer for playback by the port.

Syntax

```
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 error values shown in the following table.

Return code

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

- IDirectMusic8::CreateMusicBuffer
- **IDirectMusicBuffer8** Interface
- IDirectMusicPort8 Interface

IDirectMusicPort8::Read

The **Read** method 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.

Syntax

```
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 there is no more data to read.

If it fails, the method can return one of the error values shown in the following table.

Return code

E_POINTER E_NOTIMPL

Requirements

Header: Declared in dmusicc.h.

See Also

• Capturing MIDI

• IDirectMusicPort8 Interface

IDirectMusicPort8::SetChannelPriority

The **SetChannelPriority** method sets the priority of a MIDI channel. For an overview, see <u>Channels</u>.

Syntax

```
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 error values shown in the following table.

Return code

E_FAIL

E_INVALIDARG E_OUTOFMEMORY E_NOTIMPL

Requirements

Header: Declared in dmusicc.h.

See Also

- <u>DMUS_CHANNEL_PRIORITY_PMSG</u>
- **IDirectMusicPort8** Interface
- IDirectMusicPort8::GetChannelPriority

IDirectMusicPort8::SetDirectSound

The **SetDirectSound** method overrides the default DirectSound device object or buffer, or both, to which a port's waveform data is streamed. This method is also used to disconnect the port from DirectSound.

Syntax

```
HRESULT SetDirectSound(
   LPDIRECTSOUND pDirectSound,
   LPDIRECTSOUNDBUFFER pDirectSoundBuffer
);
```

Parameters

pDirectSound

Address of the **IDirectSound8** interface of the DirectSound device object to which the port is to be connected, or NULL to disconnect and release the existing DirectSound device 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 error values shown in the following table.

Return codeDMUS_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** Interface
- **IDirectMusicPort8::Activate**
- IDirectMusicPort8::GetFormat

IDirectMusicPort8::SetNumChannelGroups

The **SetNumChannelGroups** method changes the number of channel groups that the application needs on the port.

Syntax

```
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 error values shown in the following table.

Return code

E_FAIL E_INVALIDARG E_NOTIMPL E_OUTOFMEMORY

Requirements

Header: Declared in dmusicc.h.

See Also

• <u>Channels</u>

- IDirectMusicPort8 Interface
 IDirectMusicPort8::GetNumChannelGroups

IDirectMusicPort8::SetReadNotificationHandle

The **SetReadNotificationHandle** method 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.

Syntax

```
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 error values shown in the following table.

Return codeDMUS_E_DMUSIC_RELEASEDE_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

- <u>Capturing MIDI</u>
 <u>IDirectMusicPort8</u> Interface

IDirectMusicPort8::UnloadInstrument

The **UnloadInstrument** method unloads a previously downloaded <u>DLS</u> instrument.

Syntax

```
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 error values shown in the following table.

Return code

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

- **IDirectMusicPort8** Interface
- Working with Instruments

IDirectMusicPortDownload8 Interface

The **IDirectMusicPortDownload8** interface enables an application to communicate directly with a port that supports <u>DLS</u> downloading and to download memory chunks directly to the port. The interface is used primarily by authoring applications that edit DLS instruments directly.

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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicPortDownload8** interface exposes the following methods.

Method	Description
<u>AllocateBuffer</u>	Creates a buffer for downloading DLS data to the port.
<u>GetAppend</u>	Retrieves the amount of memory that the port needs to be appended to the end of a download buffer.
<u>GetBuffer</u>	Retrieves the pointer of a buffer whose unique identifier is known.
<u>GetDLId</u>	Obtains sequential identifiers for one or more download buffers.

Buffers

Data

on
ls a waveform or instrument
to the port.

The LPDIRECTMUSICPORTDOWNLOAD8 type is defined as a pointer to this interface.

typedef IDirectMusicPortDownload8 *LPDIRECTMUSICPORTDOWNLOAD8;

Requirements

Header: Declared in dmusicc.h.

See Also

- <u>DirectMusic Interfaces</u>
- Low-Level DLS

IDirectMusicPortDownload8::AllocateBuffer

The **AllocateBuffer** method allocates a chunk of memory for downloading <u>DLS</u> data to the port and returns an <u>IDirectMusicDownload8</u> interface pointer that allows access to this buffer.

Syntax

```
HRESULT AllocateBuffer(
   DWORD dwSize,
   IDirectMusicDownload** ppIDMDownload
);
```

Parameters

dwSize

Requested size of buffer.

ppIDMDownload

Address of a variable that receives the **<u>IDirectMusicDownload8</u>** interface pointer.

Return Values

If the method succeeds, it returns S_OK.

If it fails, the method can return one of the error values shown in the following table.

Return codeE_POINTERE_INVALIDARGE_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 Interface
- IDirectMusicPortDownload8::GetBuffer
- Low-Level DLS

IDirectMusicPortDownload8::Download

The **Download** method downloads a waveform or instrument definition to the port. The memory must first be allocated by using the **IDirectMusicPortDownload8::AllocateBuffer** method.

Syntax

```
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 error values shown in the following table.

Return code

<u>E_POINTER</u>
E_FAIL
DMUS E ALREADY DOWNLOADED
DMUS E BADARTICULATION
DMUS_E_BADINSTRUMENT
DMUS_E_BADOFFSETTABLE
DMUS_E_BADWAVE
DMUS_E_BADWAVELINK

DMUS_	E	<u>BUFFERNOTSET</u>
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 <u>Low-Level DLS</u>.

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

- <u>DMUS_DOWNLOADINFO</u>
- <u>DMUS_OFFSETTABLE</u>
- IDirectMusicPortDownload8 Interface
- IDirectMusicPortDownload8::Unload

IDirectMusicPortDownload8::GetAppend

The **GetAppend** method 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.

Syntax

```
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 WAV 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 error values shown in the following table.

Return code

E_POINTER E_NOTIMPL

Requirements

Header: Declared in dmusicc.h.

See Also

• IDirectMusicPortDownload8 Interface

IDirectMusicPortDownload8::GetBuffer

The **GetBuffer** method retrieves the **IDirectMusicDownload8** interface pointer of a buffer whose unique identifier is known.

Syntax

```
HRESULT GetBuffer(
   DWORD dwDLId,
   IDirectMusicDownload** ppIDMDownload
);
```

Parameters

dwDLId

Download identifier of the buffer. See **<u>DMUS_DOWNLOADINFO</u>**.

ppIDMDownload

Address of a variable that receives the **<u>IDirectMusicDownload8</u>** 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 error values shown in the following table.

Return codeE_POINTERDMUS_E_INVALID_DOWNLOADIDDMUS_E_NOT_DOWNLOADED_TO_PORT

Requirements

Header: Declared in dmusicc.h.

See Also

- IDirectMusicPortDownload8 Interface
- <u>IDirectMusicDownload8::GetBuffer</u>
- <u>IDirectMusicPortDownload8::GetDLId</u>
- Low-Level DLS

IDirectMusicPortDownload8::GetDLId

The **GetDLId** method 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.

Syntax

```
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 error values shown in the following table.

Return code

E_POINTER E_INVALIDARG

Requirements

Header: Declared in dmusicc.h.

See Also

- IDirectMusicPortDownload8 Interface
- <u>IDirectMusicPortDownload8::GetBuffer</u>
- Low-Level DLS

IDirectMusicPortDownload8::Unload

The **Unload** method unloads a buffer that was previously downloaded by using **IDirectMusicPortDownload8::Download**.

Syntax

```
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:

Return code

E_NOINTERFACE DMUS_E_SYNTHNOTCONFIGURED

Requirements

Header: Declared in dmusicc.h.

See Also

• IDirectMusicPortDownload8 Interface

IDirectMusicScript8 Interface

The **IDirectMusicScript8** interface represents a script containing 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.

Method Description **CallRoutine** Executes a routine in the script. Retrieves the name of a routine in a **EnumRoutine** script. Retrieves the name of a variable in a **EnumVariable** script. Retrieves a 32-bit signed value from a GetVariableNumber variable declared in the script. Retrieves an object pointer from a variable declared in the script, or **GetVariableObject** retrieves an object embedded or referenced in the script file. Retrieves a variant value from a **GetVariableVariant** variable declared in the script. Associates the script with the Init performance that will play the sounds. Assigns a 32-bit signed value to a **SetVariableNumber** variable declared in the script. Assigns an object interface pointer to a **SetVariableObject** variable declared in the script. Assigns a variant value to a variable

In addition to the methods inherited from **IUnknown**, the **IDirectMusic8** interface exposes the following methods.

SetVariableVariant

Requirements

Header: Declared in dmusici.h.

See Also

See Also

- <u>DirectMusic Interfaces</u>
- <u>Using Audio Scripts</u>

IDirectMusicScript8::CallRoutine

The **CallRoutine** method executes a routine in the script.

Syntax

```
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 parameter to NULL if you do not want error information.

Return Values

If the method succeeds, the return value is S_OK or <u>DMUS_S_GARBAGE_COLLECTED</u>. See <u>Garbage Collection</u>.

If the method fails, return values can include those in the following table.

Return codeDMUS_E_AUDIOVBSCRIPT_OPERATIONFAILUREDMUS_E_AUDIOVBSCRIPT_RUNTIMEERRORDMUS_E_AUDIOVBSCRIPT_SYNTAXERRORDMUS_E_NOT_INITDMUS_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.

See Also

- IDirectMusicScript8 Interface
- <u>Using Audio Scripts</u>

IDirectMusicScript8::EnumRoutine

The **EnumRoutine** method retrieves the name of a routine in a script. This method might be used by authoring applications that need to enumerate all routines in a script.

Syntax

```
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:

Return code	Description
S_OK	The routine was enumerated.
S_FALSE	There is no routine with the supplied index value.
DMUS_S_GARBAGE_COLLECTED	See <u>Garbage Collection</u> .
DMUS_S_STRING_TRUNCATED	The name is longer than MAX_PATH.

If the method fails, return values can include the following:

Return codeDMUS_E_NOT_INITE_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicScript8** Interface
- IDirectMusicScript8::EnumVariable
- <u>Using Audio Scripts</u>

IDirectMusicScript8::EnumVariable

The **EnumVariable** method retrieves the name of a variable in a script. This method might be used by authoring applications that need to enumerate all variables in a script.

Syntax

```
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:

Return code	Description
S_OK	The routine was enumerated.
S_FALSE	There is no routine with the supplied index value.
DMUS_S_GARBAGE_COLLECTED	See <u>Garbage Collection</u> .
DMUS_S_STRING_TRUNCATED	The name is longer than MAX_PATH.

If the method fails, return values can include the following:

Return code

DMUS_E_NOT_INIT

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicScript8** Interface
- <u>IDirectMusicScript8::EnumRoutine</u>
- <u>Using Audio Scripts</u>

IDirectMusicScript8::GetVariableNumber

The **GetVariableNumber** method retrieves a 32-bit signed value from a variable declared in the script.

Syntax

```
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:

Return code	Description
S_OK	The value was retrieved.
S_FALSE	The variable does not exist in the script.

<u>DMUS_S_GARBAGE_COLLECTED</u> See <u>Garbage Collection</u>.

If the method fails, return values can include the following:

Return code

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** Interface
- IDirectMusicScript8::SetVariableNumber
- <u>Using Audio Scripts</u>

IDirectMusicScript8::GetVariableObject

The **GetVariableObject** method retrieves an object pointer from a variable declared in the script, or retrieves an object embedded or referenced in the script file.

Syntax

```
HRESULT GetVariableObject(
    WCHAR* pwszVariableName,
    REFIID riid,
    LPVOID FAR* ppv,
    DMUS_SCRIPT_ERRORINFO* pErrInfo
);
```

Parameters

pwszVariableName

Name of the script variable or of the object referenced or embedded in the script.

riid

Unique identifier of the interface. See <u>DirectMusic Interface GUIDs</u>.

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:

Return code	Description
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:

Return code

DMUS_E_NOT_INIT
DMUS_E_SCRIPT_VARIABLE_NOT_FOUND
E_NOINTERFACE
E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicScript8 Interface
- IDirectMusicScript8::SetVariableObject
- Using Audio Scripts

IDirectMusicScript8::GetVariableVariant

The **GetVariableVariant** method retrieves a variant value from a variable declared in the script.

Syntax

```
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:

Return code	Description
S_OK	The value was retrieved.
S_FALSE	The variable does not exist in the script.

<u>DMUS_S_GARBAGE_COLLECTED</u> See <u>Garbage Collection</u>.

If the method fails, return values can include the following:

Return code

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** Interface
- IDirectMusicScript8::SetVariableVariant
- Using Audio Scripts

IDirectMusicScript8::Init

The **Init** method associates the script with the performance that will play the sounds.

Syntax

```
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:

Return code	Description
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:

Return code

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

See Also

- IDirectMusicScript8 Interface
- <u>Using Audio Scripts</u>

IDirectMusicScript8::SetVariableNumber

The **SetVariableNumber** method assigns a 32-bit signed value to a variable declared in the script.

Syntax

```
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:

Return code	Description
S_OK	The value was set.
S_FALSE	The variable does not exist in the script.

<u>DMUS_S_GARBAGE_COLLECTED</u> See <u>Garbage Collection</u>.

If the method fails, return values can include the following:

Return code

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 Interface
- IDirectMusicScript8::GetVariableNumber
- <u>Using Audio Scripts</u>

IDirectMusicScript8::SetVariableObject

The **SetVariableObject** method assigns an object interface pointer to a variable declared in the script.

Syntax

```
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:

```
Return code
```

Description

S_OK	The value was set.
S_FALSE	The variable does not exist in the script.
	1
DMUS_S_GARBAGE_COLLECTED	See <u>Garbage Collection</u> .

If the method fails, return values can include the following:

Return code

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 Interface
- IDirectMusicScript8::GetVariableObject
- <u>Using Audio Scripts</u>

IDirectMusicScript8::SetVariableVariant

The **SetVariableVariant** method assigns a variant value to a variable declared in the script.

Syntax

```
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:

Return code	Description
S_OK	The value was set.
S_FALSE	The variable does not exist in the script.
DMUS_S_GARBAGE_COLLECTED	See <u>Garbage Collection</u> .

If the method fails, return values can include the following:

Return code

<u>DMUS_E_NOT_INIT</u>
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** Interface
- IDirectMusicScript8::GetVariableObject
- <u>Using Audio Scripts</u>

IDirectMusicSegment8 Interface

The **IDirectMusicSegment8** interface represents a segment, which is a playable unit of data made up of multiple tracks.

The segment object also supports the **<u>IDirectMusicObject8</u>** and **IPersistStream** interfaces for loading its data.

IDirectMusicSegment8 supersedes the **IDirectMusicSegment** interface and introduces new methods.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicSegment8** interface exposes the following methods, arranged by category.

Instrument data

Method	Description
<u>Download</u>	Downloads band data to a performance or audiopath.
<u>Unload</u>	Unloads instrument data from a performance or audiopath.

Notification

Method	Description
AddNotificationType	Adds a type of event for which notifications are required.
<u>RemoveNotificationType</u>	Removes a a type of event for which notifications are required.

Timing and looping

Method	Description	
--------	-------------	--

<u>GetDefaultResolution</u>	Retrieves the default resolution of the segment. This is the value used to determine how times such as the segment start time are adjusted for synchronization with the rhythm.
GetLength	Retrieves the length of the segment.
GetLoopPoints	Retrieves the start and end loop points.
<u>GetRepeats</u>	Retrieves the number of times the looping portion of the segment is set to repeat.
<u>GetStartPoint</u>	Retrieves the point within the segment at which it starts playing.
SetDefaultResolution	Sets the default resolution of the segment.
<u>SetLength</u>	Sets the length, in music time, of the segment.
SetLoopPoints	Sets the start and end points of the part of the segment that repeats.
<u>SetRepeats</u>	Sets the number of times the looping portion of the segment is to repeat.
<u>SetStartPoint</u>	Sets the point within the segment at which it starts playing.

Toolgraphs

Method	Description
<u>GetGraph</u>	Retrieves the segment's toolgraph.
<u>SetGraph</u>	Assigns a toolgraph to the segment.

Track parameters

Method	Description
<u>GetParam</u>	Retrieves data from a track inside this
	segment.

Tracks

Method	Description
<u>GetTrack</u>	Searches the list of tracks for the one with the specified type, group, and index, and retrieves an interface to the track.
<u>GetTrackGroup</u>	Retrieves the group bits set on a track inside the segment.
<u>InsertTrack</u>	Inserts the specified track into the segment's list of tracks.
<u>RemoveTrack</u>	Removes a track from the segment's track list.
<u>SetTrackConfig</u>	Sets the onfiguration of a track for miscellaneous behaviors.

Miscellaneous

Method	Description
Clone	Copies all or part of the segment.
<u>Compose</u>	Composes all tracks flagged as DMUS_TRACKCONFIG_COMPOSING and places the composed tracks in this segment or in a copy of this segment.
GetAudioPathConfig	Retrieves an object that represents an audiopath configuration embedded in the segment.
InitPlay	Initializes the play state.
SetPChannelsUsed	Sets the <u>performance channels</u> that this segment uses.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Interfaces</u>
- <u>Using Segments</u>

IDirectMusicSegment8::AddNotificationType

The **AddNotificationType** method adds a type of event for which notifications are required. This method is called by the **IDirectMusicPerformance8::AddNotificationType** method, allowing the segment to generate notifications. The segment calls each track's **IDirectMusicTrack8::AddNotificationType** method.

Syntax

```
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, it returns S_OK, or S_FALSE if the type was already added.

If it fails, the method can return one of the error values shown in the following table.

Return code

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

- **IDirectMusicSegment8** Interface
- Notification and Event Handling

IDirectMusicSegment8::Clone

The **Clone** method copies all or part of the segment.

Syntax

```
HRESULT Clone(
    MUSIC_TIME mtStart,
    MUSIC_TIME mtEnd,
    IDirectMusicSegment** ppSegment
);
```

Parameters

mtStart

Start of the part to copy. If this value is less than 0 or greater than the length of the segment, the segment is copied from the beginning.

mtEnd

End time of the part to copy. If this value is past the end of the segment, or 0, or less than *mtStart*, the segment is copied 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 error values shown in the following table.

Return code 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.

For <u>style</u>-based segments, if *mtStart* is greater than 0, it should be on a measure boundary.

Requirements

Header: Declared in dmusici.h.

See Also

• **IDirectMusicSegment8** Interface

IDirectMusicSegment8::Compose

The **Compose** method composes all tracks flagged as DMUS_TRACKCONFIG_COMPOSING and places the composed tracks in this segment or in a copy of this segment.

Syntax

```
HRESULT Compose(
    MUSIC_TIME mtTime,
    IDirectMusicSegment* pFromSegment,
    IDirectMusicSegment* pToSegment,
    IDirectMusicSegment** ppComposedSegment
);
```

Parameters

mtTime

Value of type MUSIC_TIME that specifies the 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 error values shown in the following table.

Return code 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 copy 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

- IDirectMusicSegment8 Interface
- <u>IDirectMusicSegment8::SetTrackConfig</u>
- IDirectMusicTrack8::Compose
- Track Composition

IDirectMusicSegment8::Download

The **Download** method downloads band data to a performance or audiopath.

Syntax

```
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 or DMUS_S_PARTIALDOWNLOAD. See Remarks for IDirectMusicBand8::Download.

If it fails, the method may return one of the error values shown in the following table.

Return codeDMUS_E_NOT_FOUNDDMUS_E_TRACK_NOT_FOUNDE_POINTER

Remarks

All bands and waveform data in the segment are downloaded.

Always call **IDirectMusicSegment8::Unload** before releasing the segment.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegment8 Interface
- <u>Using Bands</u>

IDirectMusicSegment8::GetAudioPathConfig

The **GetAudioPathConfig** method retrieves an object that represents an audiopath configuration embedded in the segment. The object can be passed to **IDirectMusicPerformance8::CreateAudioPath**.

Syntax

```
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 error values shown in the following table.

Return code

DMUS_E_NO_AUDIOPATH_CONFIG E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

• **IDirectMusicSegment8** Interface

• <u>Creating Audiopaths</u>

IDirectMusicSegment8::GetDefaultResolution

The **GetDefaultResolution** method retrieves the default resolution of the segment. This is the value used to determine how times such as the segment start time are adjusted for synchronization with the rhythm.

Syntax

```
HRESULT GetDefaultResolution(
   DWORD* pdwResolution
);
```

Parameters

pdwResolution

Address of a variable that receives the default resolution. This value can be 0 or one of the members of the **DMUS_SEGF_FLAGS** enumeration shown in the following table.

Value	Description
DMUS_SEGF_MEASURE	Resolve times to next measure.
DMUS_SEGF_BEAT	Resolve times to next beat.
DMUS_SEGF_GRID	Resolve times to next grid.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return **<u>E_POINTER</u>**.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegment8 Interface
 IDirectMusicSegment8::SetDefaultResolution
 Segment Timing

IDirectMusicSegment8::GetGraph

The **GetGraph** method retrieves the segment's toolgraph.

Syntax

```
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 error values shown in the following table.

Return code 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.

The segment object implements **IDirectMusicGraph** directly This interface is used primarily to call the **IDirectMusicGraph8::StampPMsg** method directly on the segment. This has nothing to do with a graph object that might be

embedded in the segment. If you want to access an embedded object, you are accessing a different **IDirectMusicGraph** interface because it is an interface on the embedded graph object, not the segment itself. To obtain an interface to an embedded audiopath, use the **IDirectMusicAudioPath8::GetObjectInPath** or **IDirectMusicSegmentState8::GetObjectInPath** method.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicSegment8** Interface
- IDirectMusicSegment8::SetGraph
- <u>DirectMusic Tools</u>

IDirectMusicSegment8::GetLength

The **GetLength** method retrieves the length of the segment.

Syntax

```
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 <u>E_POINTER</u>.

Remarks

The method always returns 1 in **pmtLength* for segments created from WAV files.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicSegment8** Interface
- <u>IDirectMusicSegment8::SetLength</u>

IDirectMusicSegment8::GetLoopPoints

The **GetLoopPoints** method retrieves the start and end points of the part of the segment that repeats the number of times set by the **IDirectMusicSegment8::SetRepeats** method.

Syntax

```
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 <u>**E**POINTER</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicSegment8** Interface
- IDirectMusicSegment8::SetLoopPoints

• <u>Segment Timing</u>

IDirectMusicSegment8::GetParam

The **GetParam** method retrieves data from a track inside this segment.

Syntax

```
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 <u>Standard Track Parameters</u>.

dwGroupBits

Group that the desired track is in. Use 0xFFFFFFF for all groups. For more information, see <u>Identifying the Track</u>.

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 error values shown in the following table.

Return code

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 <u>Control Segments</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegment8 Interface
- IDirectMusicPerformance8::GetParam
- <u>IDirectMusicSegment8::SetParam</u>
- IDirectMusicTrack8::GetParamEx
- <u>Performance Parameters</u>

IDirectMusicSegment8::GetRepeats

The **GetRepeats** method retrieves the number of times the looping portion of the segment is set to repeat.

Syntax

```
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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicSegment8** Interface
- IDirectMusicSegment8::SetRepeats

IDirectMusicSegment8::GetStartPoint

The **GetStartPoint** method retrieves the point within the segment at which it starts playing.

Syntax

```
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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicSegment8** Interface
- IDirectMusicSegment8::SetStartPoint
- <u>Segment Timing</u>

IDirectMusicSegment8::GetTrack

The **GetTrack** method searches the list of tracks for the one with the specified type, group, and index, and retrieves an interface to the track object.

Syntax

```
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 <u>Standard</u> <u>Track Types</u>.

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 0xFFFFFFF.

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 error values shown in the following table.

Return code DMUS_E_NOT_FOUND E_FAIL E_POINTER

Remarks

To enumerate all tracks, use GUID_NULL for the *rguidType* and 0xFFFFFFF 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 <u>Identifying the Track</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicSegment8** Interface
- IDirectMusicSegment8::InsertTrack
- DirectMusic Tracks

IDirectMusicSegment8::GetTrackGroup

The **GetTrackGroup** method retrieves the group bits set on a track inside the segment.

Syntax

```
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 the **DWORD** 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 error values shown in the following table.

Return code DMUS_E_NOT_FOUND E_INVALIDARG

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegment8 Interface
- <u>IDirectMusicSegment8::InsertTrack</u>
- Identifying the Track

IDirectMusicSegment8::InitPlay

The **InitPlay** method initializes the play state. This method was for internal use and is not implemented in versions later than DirectX 7.0.

Syntax

```
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 error values shown in the following table.

Return code

E_POINTER E_OUTOFMEMORY

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicSegment8 Interface

IDirectMusicSegment8::InsertTrack

The **InsertTrack** method inserts the specified track into the segment's list of tracks.

Syntax

```
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 error values shown in the following table.

```
Return code

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 <u>Identifying the Track</u>.

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 Interface
- IDirectMusicSegment8::GetTrackGroup
- <u>IDirectMusicSegment8::RemoveTrack</u>

IDirectMusicSegment8::RemoveNotificationType

The **RemoveNotificationType** method removes a type of data for which notifications are required. This method is called by the **IDirectMusicPerformance8::RemoveNotificationType** method, allowing the segment to remove notifications. The segment calls each track's **IDirectMusicTrack8::RemoveNotificationType** method.

Syntax

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 **<u>E_POINTER</u>**.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicSegment8** Interface
- Notification and Event Handling

IDirectMusicSegment8::RemoveTrack

The **RemoveTrack** method removes a track from the segment's track list.

Syntax

```
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 <u>**E**</u> <u>**POINTER**</u>.

Remarks

The track is released when removed.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegment8 Interface
- IDirectMusicSegment8::InsertTrack

IDirectMusicSegment8::SetDefaultResolution

The **SetDefaultResolution** method sets the default resolution of the segment. This is the value used to determine how times such as the segment start time are adjusted for synchronization with the rhythm.

Syntax

```
HRESULT SetDefaultResolution(
   DWORD dwResolution
);
```

Parameters

dwResolution

Default resolution. This value can be 0 or one of the members of the **DMUS_SEGF_FLAGS** enumeration shown in the following table.

Value	Description
DMUS_SEGF_MEASURE	Resolve times to next measure.
DMUS_SEGF_BEAT	Resolve times to next beat.
DMUS_SEGF_GRID	Resolve times to next grid.

Return Values

The method returns S_OK.

Remarks

This method is used primarily by secondary segments to specify whether they are synchronized to the measure, beat, or grid of the primary segment by default.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegment8 Interface
- IDirectMusicSegment8::GetDefaultResolution
- <u>Segment Timing</u>

IDirectMusicSegment8::SetGraph

The **SetGraph** method assigns a toolgraph to the segment.

Syntax

```
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 <u>E_POINTER</u>.

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

- **IDirectMusicSegment8** Interface
- <u>IDirectMusicPerformance8::SetGraph</u>
- <u>DirectMusic Tools</u>

IDirectMusicSegment8::SetLength

The **SetLength** method 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.

Syntax

```
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 error values shown in the following table.

Return codeE_INVALIDARGDMUS_E_OUT_OF_RANGE

Remarks

The length does not affect the time for which the segment plays, but it is used to determine when the segment ends for purposes of synchronization; for example, when another segment is cued with the DMUS_SEGF_QUEUE flag.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegment8 Interface
- IDirectMusicSegment8::GetLength

IDirectMusicSegment8::SetLoopPoints

The **SetLoopPoints** method sets the start and end points of the part of the segment that repeats the number of times set by the **IDirectMusicSegment8::SetRepeats** method.

Syntax

```
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 <u>DMUS_E_OUT_OF_RANGE</u>.

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** Interface
- IDirectMusicSegment8::GetLoopPoints
- <u>Segment Timing</u>

IDirectMusicSegment8::SetParam

The **SetParam** method sets data on a track inside this segment.

Syntax

```
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 <u>Standard Track</u> <u>Parameters</u>.

dwGroupBits

Group that the desired track is in. Use 0xFFFFFFF for all groups. For more information, see <u>Identifying the Track</u>.

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 for this parameter. 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 error values shown in the following table.

Return code

DMUS_E_SET_UNSUPPORTED DMUS_E_TRACK_NOT_FOUND E_POINTER

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicSegment8** Interface
- IDirectMusicPerformance8::SetParam
- IDirectMusicSegment8::GetParam
- IDirectMusicTrack8::SetParamEx
- <u>Performance Parameters</u>

IDirectMusicSegment8::SetPChannelsUsed

The **SetPChannelsUsed** method sets the <u>performance channels</u> that this segment uses. This method is usually called by a track in the <u>IDirectMusicTrack8::Init</u> method to inform the segment of which channels the track uses.

Syntax

```
HRESULT SetPChannelsUsed(
   DWORD dwNumPChannels,
   DWORD* paPChannels
);
```

Parameters

dwNumPChannels

Number of performance channels to set. This must be equal to the number of members in the array pointed to by *paPChannels*.

paPChannels

Address of an array of performance channel identifiers.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the error values shown in the following table.

Return code E_INVALIDARG E_OUTOFMEMORY E_POINTER

Remarks

This method enables the performance to ascertain 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

- **IDirectMusicSegment8** Interface
- <u>Channels</u>
- Latency and Bumper Time

IDirectMusicSegment8::SetRepeats

The **SetRepeats** method sets the number of times the looping portion of the segment is to repeat.

Syntax

```
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 Interface
- IDirectMusicSegment8::GetRepeats
- IDirectMusicSegment8::SetLoopPoints
- <u>Segment Timing</u>

IDirectMusicSegment8::SetStartPoint

The **SetStartPoint** method sets the point within the segment at which it starts playing.

Syntax

```
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 0 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 <u>DMUS E OUT OF RANGE</u>.

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 <u>Segment Timing</u>.

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 **SetStartPoint** can be called. **SetLength** is normally called by the loader. However, when a WAV file is loaded, the length is always set to 1, because a WAV file is played in clock

time and has no inherent music-time length. If you want to set the start point for a WAV file to some value other than 0, you must first call **SetLength**, setting the length to some value greater than the desired start point. The length you set does not have to be the actual length of the sound unless you intend to cue another segment after it. When the segment is played, the conversion of the start point to clock-time units is based on a tempo of 120. Alternatively, you can use DirectMusic Producer to create a segment file that has a wave track; in this case, the loader will set the correct music-time length based on the tempo of the segment.

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 Interface
- IDirectMusicSegment8::GetStartPoint
- <u>IDirectMusicSegment8::SetLength</u>
- IDirectMusicSegment8::SetLoopPoints
- IDirectMusicSegmentState8::GetStartPoint
- <u>Segment Timing</u>

IDirectMusicSegment8::SetTrackConfig

The **SetTrackConfig** method sets the configuration of a track for miscellaneous behaviors.

Syntax

```
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 <u>Standard Track Types</u>.

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 error values shown in the following table.

Return code

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:

Value	Description
DMUS_TRACKCONFIG_COMPOSING	Use this track to co
DMUS_TRACKCONFIG_CONTROL_ENABLED	Enable IDirectMu
DMUS_TRACKCONFIG_CONTROL_NOTIFICATION	When played in a c of primary segment
DMUS_TRACKCONFIG_CONTROL_PLAY	When played in a c primary segment tr
DMUS_TRACKCONFIG_DEFAULT	The combination of DMUS_TRACKC(DMUS_TRACKC(DMUS_TRACKC(
DMUS_TRACKCONFIG_FALLBACK	The track tries to generation primary and contro information.
DMUS_TRACKCONFIG_LOOP_COMPOSE	Regenerate data ea
DMUS_TRACKCONFIG_NOTIFICATION_ENABLED	Enable notification
DMUS_TRACKCONFIG_OVERRIDE_ALL	The track tries to get the control and prir
	The track tries to g

DMUS_TRACKCONFIG_OVERRIDE_PRIMARY	the primary segmer
DMUS_TRACKCONFIG_PLAY_CLOCKTIME	Play in clock time,
DMUS_TRACKCONFIG_PLAY_COMPOSE	Regenerate data ea
DMUS_TRACKCONFIG_PLAY_ENABLED	Enable track to sen
DMUS_TRACKCONFIG_TRANS1_FROMSEGCURREN	In composing trans current place in the
DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART	In composing trans start of the first seg
DMUS_TRACKCONFIG_TRANS1_TOSEGSTART	In composing trans start of the second s

The following table shows which track configuration flags are valid for standard tracks.

Track	Valid flags
	DMUS_TRACKCONFIG_CONTROL_ENABLED
	DMUS_TRACKCONFIG_CONTROL_PLAY
	DMUS_TRACKCONFIG_PLAY_CLOCKTIME
Band	DMUS_TRACKCONFIG_PLAY_ENABLED
	DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
	DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
	DMUS_TRACKCONFIG_TRANS1_TOSEGSTART
	DMUS_TRACKCONFIG_CONTROL_ENABLED
	DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
Chord	DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
CHOIU	DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
	DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
	DMUS_TRACKCONFIG_TRANS1_TOSEGSTART
	DMUS_TRACKCONFIG_CONTROL_ENABLED
Chordmap	DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
	DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
	DMUS_TRACKCONFIG_TRANS1_TOSEGSTART
	DMUS_TRACKCONFIG_CONTROL_ENABLED
	DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
	DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
Command	DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT

	DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
	DMUS TRACKCONFIG TRANS1 TOSEGSTART
	DMUS_TRACKCONFIG_PLAY_CLOCKTIME
Lyrics	DMUS_TRACKCONFIG_PLAY_ENABLED
Marker	DMUS_TRACKCONFIG_CONTROL_ENABLED
	DMUS_TRACKCONFIG_COMPOSING
	DMUS_TRACKCONFIG_CONTROL_ENABLED
	DMUS TRACKCONFIG CONTROL NOTIFICATION
N (C 1 1	DMUS TRACKCONFIG CONTROL PLAY
Melody	DMUS_TRACKCONFIG_LOOP_COMPOSE
formulation	DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
(Not implemented	DMUS_TRACKCONFIG_PLAY_COMPOSE
for this release)	DMUS_TRACKCONFIG_PLAY_ENABLED
	DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
	DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
	DMUS_TRACKCONFIG_TRANS1_TOSEGSTART
	DMUS_TRACKCONFIG_CONTROL_ENABLED
	DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
	DMUS_TRACKCONFIG_CONTROL_PLAY
	DMUS_TRACKCONFIG_FALLBACK
Motif	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
Darameter control	DMUS_TRACKCONFIG_PLAY_CLOCKTIME
Parameter control	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
	DMUS_TRACKCONFIG_PLAY_CLOCKTIME

Script	DMUS_TRACKCONFIG_PLAY_ENABLED
	DMUS_TRACKCONFIG_PLAY_CLOCKTIME
Segment Trigger	DMUS_TRACKCONFIG_PLAY_ENABLED
	DMUS_TRACKCONFIG_CONTROL_PLAY
	DMUS_TRACKCONFIG_FALLBACK
Coquerce	DMUS_TRACKCONFIG_OVERRIDE_ALL
Sequence	DMUS_TRACKCONFIG_OVERRIDE_PRIMARY
	DMUS_TRACKCONFIG_PLAY_CLOCKTIME
	DMUS_TRACKCONFIG_PLAY_ENABLED
	DMUS_TRACKCONFIG_COMPOSING
	DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
	DMUS_TRACKCONFIG_CONTROL_PLAY
	DMUS_TRACKCONFIG_LOOP_COMPOSE
Cignnest	DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
Signpost	DMUS_TRACKCONFIG_PLAY_COMPOSE
	DMUS_TRACKCONFIG_PLAY_ENABLED
	DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
	DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
	DMUS_TRACKCONFIG_TRANS1_TOSEGSTART
	DMUS_TRACKCONFIG_CONTROL_ENABLED
	DMUS_TRACKCONFIG_CONTROL_NOTIFICATION
	DMUS_TRACKCONFIG_CONTROL_PLAY
	DMUS_TRACKCONFIG_FALLBACK
	DMUS_TRACKCONFIG_NOTIFICATION_ENABLED
Style	DMUS_TRACKCONFIG_OVERRIDE_ALL
	DMUS_TRACKCONFIG_OVERRIDE_PRIMARY
	DMUS_TRACKCONFIG_PLAY_ENABLED
	DMUS_TRACKCONFIG_TRANS1_FROMSEGCURRENT
_	DMUS_TRACKCONFIG_TRANS1_FROMSEGSTART
	DMUS_TRACKCONFIG_TRANS1_TOSEGSTART
	DMUS_TRACKCONFIG_CONTROL_PLAY
Sysex	DMUS_TRACKCONFIG_PLAY_CLOCKTIME
-	DMUS_TRACKCONFIG_PLAY_ENABLED
	DMUS_TRACKCONFIG_CONTROL_ENABLED
	DMUS_TRACKCONFIG_CONTROL_PLAY
	DMUS_TRACKCONFIG_PLAY_CLOCKTIME
Tompo	DMUS_TRACKCONFIG_PLAY_ENABLED
Tempo	

	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

- **IDirectMusicSegment8** Interface
- <u>IDirectMusicSegmentState8::SetTrackConfig</u>
- <u>Self-Controlling Segments</u>
- <u>Track Configuration</u>

IDirectMusicSegment8::Unload

The **Unload** method unloads instrument data from a performance or audiopath.

Syntax

```
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 error values shown in the following table.

Return codeDMUS_E_TRACK_NOT_FOUNDE_POINTER

Remarks

The method succeeds even if no data was previously downloaded.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegment8 Interface
- IDirectMusicSegment8::Download
 Downloading and Unloading Bands

IDirectMusicSegmentState8 Interface

The **IDirectMusicSegmentState8** interface represents a playing instance of a segment. When the **IDirectMusicPerformance8::PlaySegment** or **IDirectMusicPerformance8::PlaySegmentEx** method is called, the performance creates a segment state object that tracks the state of the playing segment. This object be passed to**IDirectMusicPerformance8::StopEx**.

IDirectMusicSegmentState8 supersedes **IDirectMusicSegmentState** and adds new methods.

Method	Description
<u>GetObjectInPath</u>	Retrieves an interface for an object in the audiopath on which this segment state is playing.
<u>GetRepeats</u>	Retrieves the number of times that the looping portion of the segment is set to repeat.
<u>GetSeek</u>	Retrieves the seek pointer in the segment state.
<u>GetSegment</u>	Retrieves a pointer to the segment that owns this segment state.
<u>GetStartPoint</u>	Retrieves the time within the segment at which it started playing.
<u>GetStartTime</u>	Retrieves the performance time at which the segment started playing.
<u>SetTrackConfig</u>	Sets the configuration of a track in the parent segment.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicSegmentState8** interface exposes the following methods.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>DirectMusic Interfaces</u>

IDirectMusicSegmentState8::GetObjectInPath

The **GetObjectInPath** method retrieves an interface for an object in the audiopath on which this segment state is playing.

Syntax

```
HRESULT GetObjectInPath(
   DWORD dwPChannel,
   DWORD dwStage,
   DWORD dwBuffer,
   REFGUID guidObject,
   DWORD dwIndex,
   REFGUID iidInterface,
   Void** ppObject
);
```

Parameters

dwPChannel

<u>Performance channel</u> 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 values listed in the following table.

Value	Description
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 <u>DMO</u> in the buffer.
DMUS_PATH_MIXIN_BUFFER	A global <u>mix-in buffer</u> .

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 <u>Garbage Collection</u>.

If it fails, the method can return one of the error values shown in the following table.

Return code

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*:

Value

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

- **<u>IDirectMusicSegmentState8</u>** Interface
- IDirectMusicAudioPath8::GetObjectInPath
- <u>Retrieving Objects from an Audiopath</u>

IDirectMusicSegmentState8::GetRepeats

The **GetRepeats** method retrieves the number of times that the looping portion of the segment is set to repeat.

Syntax

```
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 <u>**E**</u> **POINTER**.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegmentState8 Interface
- IDirectMusicSegment8::SetRepeats

IDirectMusicSegmentState8::GetSeek

The **GetSeek** method 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.

Syntax

```
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 **<u>E_POINTER</u>**.

Remarks

This method is not an accurate way of determining the current play position, because it seeks beyond the prepare time. For a better way to determine the current play position, see <u>Pausing Segments</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegmentState8 Interface
 Prepare Time

IDirectMusicSegmentState8::GetSegment

The **GetSegment** method retrieves a pointer to the segment that generated this segment state.

Syntax

```
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 error values shown in the following table.

Return code DMUS_E_NOT_FOUND E_POINTER

Remarks

The pointer returned in *ppSegment* must be released by the application.

Requirements

Header: Declared in dmusici.h.

See Also

• **<u>IDirectMusicSegmentState8</u>** Interface

IDirectMusicSegmentState8::GetStartPoint

The **GetStartPoint** method retrieves the time within the segment at which it started playing.

Syntax

```
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 play will begin or began.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return **<u>E POINTER</u>**.

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 <u>Segment Timing</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicSegmentState8 Interface

- <u>IDirectMusicSegment8::SetStartPoint</u>
- IDirectMusicSegmentState8::GetStartTime
 Segment Timing

IDirectMusicSegmentState8::GetStartTime

The **GetStartTime** method retrieves the performance time at which the segment started playing.

Syntax

```
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 <u>E_POINTER</u>.

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 <u>Segment Timing</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicSegmentState8 Interface

- IDirectMusicSegment8::GetStartPoint
- IDirectMusicSegment8::SetStartPoint
- IDirectMusicSegmentState8::GetStartPoint

IDirectMusicSegmentState8::SetTrackConfig

The **SetTrackConfig** method sets the configuration of a track in the parent segment.

Syntax

```
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 <u>Standard Track Types</u>.

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 error values shown in the following table.

Return codeDMUS_E_TRACK_NOT_FOUNDE_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

- **IDirectMusicSegmentState8** Interface
- IDirectMusicSegment8::SetTrackConfig
- <u>Self-Controlling Segments</u>
- <u>Track Configuration</u>

IDirectMusicStyle8 Interface

The **IDirectMusicStyle8** interface represents a collection of <u>patterns</u>, motifs, and bands used in run-time composition of musical segments. For an overview, see <u>Using Styles</u>.

Style objects also support the **IDirectMusicObject8** and **IPersistStream** interfaces for loading their data.

IDirectMusicStyle8 supersedes the **IDirectMusicStyle** interface and provides a new method, **EnumPattern**.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicStyle8** interface exposes the following methods, arranged by category.

Data retrieval

Method	Description
GetBand	Retrieves a band specified by name.
<u>GetChordMap</u>	Retrieves a <u>chordmap</u> specified by
	name.
GetDefaultBand	Retrieves the style's default band.
GetDefaultChordMap	Retrieves the style's default chordmap.
<u>GetEmbellishmentLength</u>	Finds the shortest and longest lengths
	for <u>patterns</u> of the specified
	embellishment type and <u>groove level</u> .
<u>GetMotif</u>	Creates a segment containing a motif
	specified by name.
<u>GetTempo</u>	Retrieves the recommended tempo of
	the style.
GetTimeSignature	Retrieves the style's time signature.

Enumeration

Method

Description

EnumBand	Retrieves the name of the band with a specified index value.
EnumChordMap	Retrieves the name of the chordmap with a specified index value.
<u>EnumMotif</u>	Retrieves the name of a motif with a specified index value.
<u>EnumPattern</u>	Retrieves the name of a pattern with a specified index value and type.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>DirectMusic Interfaces</u>

IDirectMusicStyle8::EnumBand

The **EnumBand** method retrieves the name of the band with a specified index value.

Syntax

```
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 <u>DMUS_S_STRING_TRUNCATED</u> if the length of the name is greater than MAX_PATH.

If it fails, the method can return one of the error values shown in the following table.

Return code <u>DMUS_E_TYPE_UNSUPPORTED</u> <u>E_POINTER</u>

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicStyle8 Interface
- <u>Using Bands</u>

IDirectMusicStyle8::EnumChordMap

The **EnumChordMap** method retrieves the name of the <u>chordmap</u> with a specified index value.

Syntax

```
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 <u>DMUS_S_STRING_TRUNCATED</u> if the length of the name is greater than MAX_PATH.

If it fails, the method can return one of the error values shown in the following table.

Return code <u>DMUS E TYPE UNSUPPORTED</u> <u>E_POINTER</u>

Remarks

DirectMusic Producer does not support embedding chordmap references in style files.

Requirements

Header: Declared in dmusici.h.

See Also

• **IDirectMusicStyle8** Interface

IDirectMusicStyle8::EnumMotif

The **EnumMotif** method retrieves the name of a motif with a specified index value.

Syntax

```
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 <u>DMUS_S_STRING_TRUNCATED</u> if the length of the motif name is greater than MAX_PATH.

If it fails, the method can return <u>E_POINTER</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicStyle8 Interface

- IDirectMusicStyle8::GetMotif
 Using Motifs

IDirectMusicStyle8::EnumPattern

The **EnumPattern** method retrieves the name of a <u>pattern</u> with a specified index value and type. The name can be passed to the <u>IDirectMusicPatternTrack8::SetPatternByName</u> method.

Syntax

```
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. The buffer 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 error values shown in the following table.

Return codeE_INVALIDARGE_POINTER.

Requirements

Header: Declared in dmusici.h.

See Also

• **IDirectMusicStyle8** Interface

IDirectMusicStyle8::GetBand

The **GetBand** method retrieves a band specified by name.

Syntax

```
HRESULT GetBand(
    WCHAR* pwszName,
    IDirectMusicBand** ppBand
);
```

Parameters

pwszName

Name of the band to retrieve. This name is created by the author of the style.

ppBand

Address of a variable that receives the **<u>IDirectMusicBand8</u>** interface.

Return Values

If the method succeeds, the return value is S_OK if a band is returned, or <u>S_FALSE</u> if there is no band with that name.

If the method fails, the return value can be <u>**E_POINTER</u>**.</u>

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicStyle8 Interface
- IDirectMusicStyle8::GetDefaultBand
- <u>Using Bands</u>

IDirectMusicStyle8::GetChordMap

The **GetChordMap** method retrieves a <u>chordmap</u> specified by name.

Syntax

```
HRESULT GetChordMap(
    WCHAR* pwszName,
    IDirectMusicChordMap** ppChordMap
);
```

Parameters

pwszName

Name of the chordmap to retrieve. This name is created by the author of the style.

ppChordMap

Address of a variable that receives a pointer to the **<u>IDirectMusicChordMap8</u>** interface.

Return Values

If the method succeeds, the return value is S_OK if a chordmap is returned, or <u>S_FALSE</u> if there is no chordmap by that name.

If *ppChordMap* is not a valid pointer, the method returns <u>E_POINTER</u>.

Remarks

DirectMusic Producer does not support embedding chordmap references in style files.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicStyle8** Interface
- IDirectMusicStyle8::GetDefaultChordMap
 Using Chordmaps

IDirectMusicStyle8::GetDefaultBand

The **GetDefaultBand** method retrieves the style's default band.

Syntax

```
HRESULT GetDefaultBand(
    IDirectMusicBand ** ppBand
);
```

Parameters

ppBand

Address of a variable that receives the **<u>IDirectMusicBand8</u>** 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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- **IDirectMusicStyle8** Interface
- IDirectMusicStyle8::GetBand
- <u>Using Bands</u>

IDirectMusicStyle8::GetDefaultChordMap

The **GetDefaultChordMap** method retrieves the style's default <u>chordmap</u>.

Syntax

```
HRESULT GetDefaultChordMap(
    IDirectMusicChordMap** ppChordMap
);
```

Parameters

ppChordMap

Address of a variable that receives a pointer to the **<u>IDirectMusicChordMap8</u>** 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 <u>E_POINTER</u>.

Remarks

DirectMusic Producer does not support embedding chordmap references in style files.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicStyle8 Interface
- IDirectMusicStyle8::GetChordMap

• <u>Using Chordmaps</u>

IDirectMusicStyle8::GetEmbellishmentLength

The **GetEmbellishmentLength** method finds the shortest and longest lengths for <u>patterns</u> of the specified embellishment type and <u>groove level</u>.

Syntax

```
HRESULT GetEmbellishmentLength(
   DWORD dwType,
   DWORD dwLevel,
   DWORD* pdwMin,
   DWORD* pdwMax
);
```

Parameters

dwType

Embellishment type. This can be one of the **DMUS_COMMANDT_TYPES** enumeration, or a value defined in DirectMusic Producer as a custom embellishment.

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 <u>S_FALSE</u>.

If it fails, the method can return <u>E_POINTER</u>.

Remarks

If there are no patterns of the specified type and groove level, the method returns S_FALSE.

Requirements

Header: Declared in dmusici.h.

See Also

• **IDirectMusicStyle8** Interface

IDirectMusicStyle8::GetMotif

The **GetMotif** method creates a segment containing a motif specified by name.

Syntax

```
HRESULT GetMotif(
    WCHAR* pwszName,
    IDirectMusicSegment** ppSegment
);
```

Parameters

pwszName

Name of the motif.

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 **<u>E_POINTER</u>**.

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 <u>pattern</u>.

If there is no motif with the name, the method returns S_FALSE.

Requirements

Header: Declared in dmusici.h.

See Also

- IDirectMusicSegment8 Interface
- **IDirectMusicStyle8** Interface
- Using Motifs

IDirectMusicStyle8::GetTempo

The **GetTempo** method retrieves the recommended tempo of the style.

Syntax

```
HRESULT GetTempo(
    double* pTempo
);
```

Parameters

рТетро

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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicStyle8 Interface

IDirectMusicStyle8::GetTimeSignature

The **GetTimeSignature** method retrieves the time signature of the style.

Syntax

```
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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicStyle8 Interface

IDirectMusicSynth Interface

The **IDirectMusicSynth** interface is implemented by synthesizers. Applications do not use this interface but communicate with the port by using **IDirectMusicPort8**.

For more information, see the following topic on mdsn.microsoft.com:

• DirectMusic User Mode Synth and Synth Sink Interfaces

Requirements

Header: Declared in dmusics.h.

See Also

• <u>DirectMusic Interfaces</u>

IDirectMusicSynthSink Interface

The **IDirectMusicSynthSink** interface is implemented by a synthesizer sink, which pull data from a synthesizer. Applications do not use this interface.

A custom sink for the default DirectX 7.0 synthesizer (created by calling **IDirectMusicPerformance8::Init**) must implement this interface. Such a sink will not work with the default synthesizer for DirectX 8.0 and later (created by calling **IDirectMusicPerformance8::InitAudio**). To capture output from the later version of the synthesizer, you must implement a DMO on the primary buffer.

For more information, see the following topic on mdsn.microsoft.com:

DirectMusic User Mode Synth and Synth Sink Interfaces

Requirements

Header: Declared in dmusics.h.

See Also

• <u>DirectMusic Interfaces</u>

IDirectMusicThru8 Interface

The **IDirectMusicThru8** interface supports thruing of MIDI 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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicThru8** interface exposes the following methods.

ThruChannelEstablishes or breaks a thruing connection between a channel on a capture port and a channel on another port.	Method	Description	
1	<u>ThruChannel</u>	connection between a channel on a capture port and a channel on another	

The **LPDIRECTMUSICTHRU8** type is defined as a pointer to this interface.

typedef IDirectMusicThru8 *LPDIRECTMUSICTHRU8;

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Interfaces</u>
- <u>Capturing MIDI</u>

IDirectMusicThru8::ThruChannel

The **ThruChannel** method establishes or breaks a thruing connection between a channel on a capture port and a channel on another port.

Syntax

```
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.

dw Destination Channel Group

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 error values shown in the following table.

Return codeE_NOTIMPLE_INVALIDARGDMUS_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.

See Also

• IDirectMusicThru8 Interface

IDirectMusicTool8 Interface

The **IDirectMusicTool8** interface represents a tool object that processes messages.

Methods of this interface are implemented by tools 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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicTool8** interface exposes the following methods.

Method	Description
Clone	Creates a new instance of the tool.
Flush	Flushes messages from the queue when the performance stops.
	Retrieves the size of the array that must
<u>GetMediaTypeArraySize</u>	be passed in to the IDirectMusicTool8::GetMediaTypes
	method. Retrieves a list of the types of messages
<u>GetMediaTypes</u>	that this tool supports.
<u>GetMsgDeliveryType</u>	Retrieves the tool's delivery type, which determines when messages are to be delivered to the tool.
Init	Initializes the tool.
ProcessPMsg	Performs the main task of the tool.

Requirements

Header: Declared in dmplugin.h.

See Also

- <u>DirectMusic Interfaces</u>
- <u>DirectMusic Tools</u>
 <u>Message Creation and Delivery</u>

IDirectMusicTool8::Clone

The **Clone** method creates a new instance of the tool.

Syntax

```
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 <u>**E_POINTER</u>** or <u>**E_OUTOFMEMORY**</u>.</u>

Requirements

Header: Declared in dmplugin.h.

See Also

• IDirectMusicTool8 Interface

IDirectMusicTool8::Flush

The **Flush** method 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.

Syntax

```
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:

Return code DMUS_S_REQUEUE DMUS_S_FREE S_OK

If it fails, the method can return **<u>E_POINTER</u>**.

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 message is placed back in the queue. The tool can put a new time stamp and parameters on the message, or change the delivery type.

If the return value is DMUS_S_FREE, the message is freed by the performance.

If the return value is S_OK, the message is not freed by the performance. The tool might be holding onto the message for some reason, or the tool might already have freed the message.

Be sure not to create a circular reference to the performance represented by *pPerf*. For more information, see <u>DirectMusic Tools</u>.

Requirements

Header: Declared in dmplugin.h.

See Also

• **IDirectMusicTool8** Interface

IDirectMusicTool8::GetMediaTypeArraySize

The **GetMediaTypeArraySize** method 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**.

Syntax

```
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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmplugin.h.

See Also

• IDirectMusicTool8 Interface

IDirectMusicTool8::GetMediaTypes

The **GetMediaTypes** method retrieves a list of the types of messages that this tool supports.

Syntax

```
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 error values shown in the following table.

Return code

<u>E_POINTER</u> E_INVALIDARG

E_NOTIMPL

Remarks

If the method returns E_NOTIMPL, the tool processes all media types.

Requirements

Header: Declared in dmplugin.h.

See Also

• IDirectMusicTool8 Interface

IDirectMusicTool8::GetMsgDeliveryType

The **GetMsgDeliveryType** method retrieves the tool's delivery type, which determines when messages are to be delivered to the tool.

Syntax

```
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 <u>E_POINTER</u>.

Remarks

For an overview of the delivery mechanism, see <u>Message Creation and Delivery</u>.

Requirements

Header: Declared in dmplugin.h.

See Also

• **IDirectMusicTool8** Interface

IDirectMusicTool8::Init

The **Init** method 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.

Syntax

```
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 error values shown in the following table.

Return code

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 <u>DirectMusic Tools</u>.

Requirements

Header: Declared in dmplugin.h.

See Also

- IDirectMusicGraph8::InsertTool
 IDirectMusicTool8 Interface

IDirectMusicTool8::ProcessPMsg

The **ProcessPMsg** method 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**.

Syntax

```
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:

Return codeDMUS_S_REQUEUEDMUS_S_FREES_OK

If it fails, the method can return **<u>E_POINTER</u>**.

Remarks

If the method returns DMUS_S_REQUEUE, the message is placed back in the queue. The tool can put a new time stamp and parameters on the message, or change the delivery type.

If the return value is DMUS_S_FREE, the message is freed by the performance.

If the return value is S_OK, the message is not freed by the performance. The tool might be holding onto the message for some reason, or the tool might already have freed the message.

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 <u>DirectMusic Tools</u>.

Requirements

Header: Declared in dmplugin.h.

See Also

- IDirectMusicTool8 Interface
- IDirectMusicPerformance8::SendPMsg
- Message Creation and Delivery
- <u>Messages</u>

IDirectMusicTrack8 Interface

The **IDirectMusicTrack8** interface represents a track object, which can store any kind of data for a segment.

The methods of this interface are typically not called directly by applications. However, if you want to install your own data playback mechanism in DirectMusic, you must create a track object to represent it. The track object must also support the **IPersistStream** interface for loading its data.

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.

In addition to the methods inherited from **IUnknown**, the **IDirectMusicTrack8** interface exposes the following methods, arranged by category.

Notification

Method	Description	
AddNotificationType	Adds a type of event for which notifications are required.	
<u>RemoveNotificationType</u>	Removes a type of event for which notifications are required.	

Parameters

Method	Description	
GetParam	Retrieves data from a track, in music	
	time.	
	Retrieves data from a track, in either	

GetParamEx	music time or reference time.	
IsParamSupported	Ascertains whether the track supports a given data type.	
<u>SetParam</u>	Sets data on a track, in music time.	
<u>SetParamEx</u>	Sets data on a track, in either clock or music time.	

Playback

Method	Description
EndPlay	Called when the object that originally called IDirectMusicTrack8::InitPlay is destroyed.
<u>InitPlay</u>	Called when a track is ready to start playing.
<u>Play</u>	Called when the object that originally called IDirectMusicTrack8::InitPlay is destroyed.
<u>PlayEx</u>	Causes the track to play in clock time.

Miscellaneous

Method	Description	
<u>Clone</u>	Makes a copy of the track.	
<u>Compose</u>	Recomposes the track based on data from a segment.	
Init	Initializes the track.	
<u>Join</u>	Appends one track to another.	

Requirements

Header: Declared in dmplugin.h.

See Also

- <u>DirectMusic Interfaces</u>
- <u>DirectMusic Tracks</u>
 <u>Setting and Retrieving Track Parameters</u>

IDirectMusicTrack8::AddNotificationType

The **AddNotificationType** method adds a type of event for which notifications are required. It is similar to, and called by, the **IDirectMusicSegment8::AddNotificationType** method.

Syntax

```
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 error values shown in the following table.

Return codeDMUS_E_NOT_INITE_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 Interface
- <u>IDirectMusicTrack8::RemoveNotificationType</u>
- Notification and Event Handling

IDirectMusicTrack8::Clone

The **Clone** method makes a copy of the track.

Syntax

```
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 error values shown in the following table.

R	eturn code	
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.

See Also

• IDirectMusicTrack8 Interface

IDirectMusicTrack8::Compose

The **Compose** method recomposes the track based on data from a segment. DirectMusic implements this method on the signpost track to compose a chord track.

Syntax

```
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 <u>Identifying the Track</u>.

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 error values shown in the following table.

Return code E_INVALIDARG E_POINTER

Requirements

Header: Declared in dmplugin.h.

See Also

- <u>IDirectMusicTrack8 Interface</u>
- <u>IDirectMusicSegment8::Compose</u>

IDirectMusicTrack8::EndPlay

The **EndPlay** method called when the object that originally called **<u>IDirectMusicTrack8::InitPlay</u>** is destroyed.

Syntax

```
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 <u>E_POINTER</u>.

Requirements

Header: Declared in dmplugin.h.

See Also

• IDirectMusicTrack8 Interface

IDirectMusicTrack8::GetParam

The **GetParam** method retrieves data from a track, in music time.

Syntax

```
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 <u>Standard Track Parameters</u>.

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 error values shown in the following table.

Return code

DMUS_E_NOT_FOUND DMUS_E_NOT_INIT DMUS_E_TYPE_DISABLED DMUS_E_GET_UNSUPPORTED E_POINTER

Remarks

The **<u>IDirectMusicTrack8::GetParamEx</u>** method can be used for greater functionality.

Requirements

Header: Declared in dmplugin.h.

See Also

- IDirectMusicTrack8 Interface
- <u>IDirectMusicPerformance8::GetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicTrack8::IsParamSupported</u>
- <u>IDirectMusicTrack8::SetParam</u>
- <u>Performance Parameters</u>

IDirectMusicTrack8::GetParamEx

The **GetParamEx** method retrieves data from a track, in either music time or reference time.

Syntax

```
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 <u>Standard Track Parameters</u>.

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 0, 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 0 or the following flag.

ValueDescriptionDMUS_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 error values shown in the following table.

Return code

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 Interface

- IDirectMusicTrack8::GetParam
 IDirectMusicTrack8::SetParamEx

IDirectMusicTrack8::Init

The **Init** method initializes the track. This method is called by a segment when a track is added.

Syntax

```
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 error values shown in the following table.

Return code

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.

See Also

• IDirectMusicTrack8 Interface

IDirectMusicTrack8::InitPlay

The **IDirectMusicTrack8::InitPlay** method is called when a track is ready to start playing. The method returns a pointer to state data.

Syntax

```
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 error values shown in the following table.

Return code

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.

See Also

• <u>IDirectMusicTrack8 Interface</u>

IDirectMusicTrack8::IsParamSupported

The **IsParamSupported** method ascertains whether the track supports a given data type.

Syntax

```
HRESULT IsParamSupported(
    REFGUID rguidType
);
```

Parameters

rguidType

Reference to (C++) or address of (C) the identifier of the type of data. See <u>Standard Track Parameters</u>.

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 error values shown in the following table.

Return code

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 Interface
- <u>IDirectMusicTrack8::GetParam</u>
- <u>IDirectMusicTrack8::GetParamEx</u>
- <u>IDirectMusicTrack8::SetParam</u>
- <u>Performance Parameters</u>

IDirectMusicTrack8::Join

The **Join** method appends one track to another.

Syntax

```
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 <u>Identifying the Track</u>.

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 error values shown in the following table.

Return codeE_POINTERE_INVALIDARGE_OUTOFMEMORY

Remarks

This method is supported by the band, chordmap, tempo, style, chord, signpost, and command tracks.

Requirements

Header: Declared in dmplugin.h.

See Also

• IDirectMusicTrack8 Interface

IDirectMusicTrack8::Play

The **Play** method 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.

Syntax

```
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 <u>DMUS_S_END</u>.

If it fails, the method can return one of the error values shown in the following table.

Return codeDMUS_E_NOT_INITE_POINTER

Remarks

If the track is empty, the method returns DMUS_S_END.

Tracks generate messages in a medium-priority thread. You can call timeconsuming 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 Interface
- <u>IDirectMusicTrack8::PlayEx</u>

IDirectMusicTrack8::PlayEx

The **PlayEx** method 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.

Syntax

```
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 0 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 <u>DMUS_S_END</u>.

If it fails, the method can return one of the error values shown in the following table.

Return codeDMUS_E_NOT_INITE_POINTER

Requirements

Header: Declared in dmplugin.h.

See Also

- IDirectMusicTrack8 Interface
- <u>IDirectMusicTrack8::Play</u>

IDirectMusicTrack8::RemoveNotificationType

The **RemoveNotificationType** method removes a type of event for which notifications are required.. It is similar to, and called by, the **IDirectMusicSegment8::RemoveNotificationType** method.

Syntax

```
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 **<u>E_NOTIMPL</u>**.

Requirements

Header: Declared in dmplugin.h.

See Also

- IDirectMusicTrack8 Interface
- IDirectMusicTrack8::AddNotificationType
- Notification and Event Handling

IDirectMusicTrack8::SetParam

The **SetParam** method sets data on a track, in music time.

Syntax

```
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 <u>Standard Track Parameters</u>.

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 for this parameter. 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 error values shown in the following table.

Return codeDMUS_E_SET_UNSUPPORTED

DMUS_E_TYPE_DISABLED E_OUTOFMEMORY E_POINTER

Remarks

The **<u>IDirectMusicTrack8::SetParamEx</u>** method can be used for greater functionality.

Requirements

Header: Declared in dmplugin.h.

See Also

- IDirectMusicTrack8 Interface
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::SetParam</u>
- IDirectMusicTrack8::GetParam
- <u>IDirectMusicTrack8::GetParamEx</u>
- <u>IDirectMusicTrack8::IsParamSupported</u>
- IDirectMusicTrack8::SetParamEx
- <u>Setting and Retrieving Track Parameters</u>

IDirectMusicTrack8::SetParamEx

The **SetParamEx** method sets data on a track, in either clock or music time.

Syntax

```
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 <u>Standard Track Parameters</u>.

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 for this parameter. 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 0 or the following flag.

ValueDescriptionDMUS_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 error values shown in the following table.

Return code

DMUS_E_SET_UNSUPPORTED DMUS_E_TYPE_DISABLED E_OUTOFMEMORY E_POINTER

Requirements

Header: Declared in dmplugin.h.

See Also

- IDirectMusicTrack8 Interface
- <u>IDirectMusicTrack8::GetParamEx</u>
- IDirectMusicTrack8::SetParam

IKsControl Interface

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.

In addition to the methods inherited from **IUnknown**, the **IKsControl** interface exposes the following methods.

Method	Description
<u>KsProperty</u>	Retrieves or sets the value of a property.
KsEvent	Not supported by DirectMusic.
KsMethod	Not supported by DirectMusic.

Requirements

Header: Declared in dmksctrl.h.

See Also

- <u>DirectMusic Interfaces</u>
- <u>Property Sets for DirectMusic Ports</u>

IKsControl::KsProperty

The **KsProperty** method retrieves or sets the value of a property. For an overview, see <u>Property Sets for DirectMusic Ports</u>.

Syntax

```
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 error values shown in the following table.

Return code

E_FAIL
E_INVALIDARG
E_NOTIMPL
E OUTOFMEMORY
<u>E_POINTER</u>
DMUS_E_UNKNOWN_PROPERTY

Requirements

Header: Declared in dmksctrl.h.

See Also

- **IKsControl** Interface
- <u>Property Sets for DirectMusic Ports</u>

IReferenceClock Interface

The **IReferenceClock** interface represents a system reference clock. The DirectMusic master clock and a port's latency clock implement this interface.

In addition to the methods inherited from **IUnknown**, the **IReferenceClock** interface exposes the following methods.

Method	Description
<u>GetTime</u>	Retrieves the current time.
<u>AdviseTime</u>	Requests an asynchronous notification that a time has elapsed.
AdvisePeriodic	Requests an asynchronous, periodic notification that a duration has elapsed.
<u>Unadvise</u>	Cancels a request for notification.

Requirements

Header: Declared in dmusicc.h.

See Also

- <u>DirectMusic Interfaces</u>
- <u>IDirectMusic8::GetMasterClock</u>
- <u>IDirectMusicPort8::GetLatencyClock</u>
- <u>Timing</u>

IReferenceClock::AdvisePeriodic

The **AdvisePeriodic** method requests an asynchronous, periodic notification that a duration has elapsed.

Syntax

```
HRESULT AdvisePeriodic(
    REFERENCE_TIME startTime,
    REFERENCE_TIME periodTime,
    HANDLE hSemaphore,
    DWORD * pdwAdviseCookie
);
```

Parameters

startTime

Time when notification should begin.

periodTime

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 error values shown in the following table.

Return code

E_FAIL E_POINTER E_INVALIDARG E_NOTIMPL

Remarks

When the time indicated by *startTime* is reached, the semaphore whose handle is set as *hSemaphore* is released. Thereafter, the semaphore is released repetitively with a period of *periodTime*.

Requirements

Header: Declared in dmusicc.h.

See Also

- IReferenceClock Interface
- IReferenceClock::Unadvise

IReferenceClock::AdviseTime

The **AdviseTime** method requests an asynchronous notification that a time has elapsed.

Syntax

```
HRESULT AdviseTime(
    REFERENCE_TIME baseTime,
    REFERENCE_TIME streamTime,
    HANDLE hEvent,
    DWORD * pdwAdviseCookie
);
```

Parameters

baseTime

Base reference time.

streamTime

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 error values shown in the following table.

Return code

E_FAIL E_POINTER E_INVALIDARG E_NOTIMPL

Remarks

When the time *baseTime* plus *streamTime* 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 Interface
- IReferenceClock::Unadvise

IReferenceClock::GetTime

The **GetTime** method retrieves the current time.

Syntax

```
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 error values shown in the following table.

Return code

E_FAIL E_POINTER E_INVALIDARG E_NOTIMPL

Requirements

Header: Declared in dmusicc.h.

See Also

• IReferenceClock Interface

IReferenceClock::Unadvise

The **Unadvise** method cancels a request for notification.

Syntax

```
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 error values shown in the following table.

Return code

E_FAIL E_POINTER E_INVALIDARG E_NOTIMPL

Requirements

Header: Declared in dmusicc.h.

See Also

• IReferenceClock Interface

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 <u>Using DirectMusic Messages</u>.

This section contains information about the following structures used to contain message information.

Structure	Description
DMUS_CHANNEL_PRIORITY_PMSG	Contains message data about a channel priority change.
DMUS_CURVE_PMSG	Contains message data for a <u>curve</u> .
DMUS_LYRIC_PMSG	Contains message data for a string.
DMUS_MIDI_PMSG	Contains data for a standard MIDI message such as a control change or pitch bend.
DMUS_NOTE_PMSG	Contains message data for a MIDI note.
DMUS_NOTIFICATION_PMSG	Contains message data for a notification.
DMUS_PATCH_PMSG	Contains message data for a MIDI program change.
DMUS_PMSG	Contains information common to all DirectMusic messages.
DMUS_SYSEX_PMSG	Contains data for a MIDI system exclusive message.
DMUS_TEMPO_PMSG	Contains data for a message that controls the performance's tempo.
DMUS_TIMESIG_PMSG	Contains data for a message that controls the time signature of the performance.
	Contains message data for a

DMUS_TRANSPOSE_PMSG	transposition.
DMUS_WAVE_PMSG	Contains message data for a wave sound.

See Also

- <u>IDirectMusicPerformance8::AllocPMsg</u>
- <u>IDirectMusicPerformance8::FreePMsg</u>
- IDirectMusicPerformance8::SendPMsg
- IDirectMusicTool8::ProcessPMsg

DMUS_CHANNEL_PRIORITY_PMS

The **DMUS_CHANNEL_PRIORITY_PMSG** structure contains message data about a channel priority change.

Syntax

typedef struct _DMUS_CHANNEL_PRIORITY_PMSG {
 DMUS_PMSG_PART
 DWORD dwChannelPriority;
} DMUS_CHANNEL_PRIORITY_PMSG;

Members

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

- <u>IDirectMusicPerformance8::SendPMsg</u>
- IDirectMusicPort8::SetChannelPriority

DMUS_CURVE_PMSG

The **DMUS_CURVE_PMSG** structure contains message data for a <u>curve</u>.

Syntax

```
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 bCurveShape;
   BYTE bCLData;
   BYTE bFlags;
   WORD wMergeIndex;
} DMUS CURVE PMSG;
```

Members

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 the duration is in milliseconds and is unaffected by a change in tempo.

mtOriginalStart

Original start time. Must be set to either zero 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.

Value	Description
DMUS_CURVET_CCCURVE	Continuous controller curve (MIDI Control Change channel voice message; status byte &HB <i>n</i> .).
DMUS_CURVET_MATCURVE	Monophonic aftertouch curve (MIDI Channel Pressure channel voice message; status byte &HD <i>n</i>).
DMUS_CURVET_PATCURVE	Polyphonic aftertouch curve (MIDI Poly Key Pressure channel voice message, status byte &HD <i>n</i>).
DMUS_CURVET_PBCURVE	Pitch-bend curve (MIDI Pitch Bend channel voice message; status byte &HE <i>n</i>).
DMUS_CURVET_RPNCURVE	RPN curve of type defined in wParamType.
DMUS_CURVET_NRPNCURVE	<u>NRPN</u> curve of type defined in wParamType .

bCurveShape

Shape of curve. This can be one of the following values.

Value	Description	
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; otherwise ignored.

bFlags

Can be zero, or one or more of the following values.

Value	Description
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 cutoff, 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 transpose, pitch bend, volume, expression, pan, filter, mod wheel, chorus, and reverb 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
- <u>Curves</u>

DMUS_LYRIC_PMSG

The **DMUS_LYRIC_PMSG** structure contains message data for a string.

Syntax

```
typedef struct _DMUS_LYRIC_PMSG {
   DMUS_PMSG_PART
   WCHAR wszString[1];
} DMUS_LYRIC_PMSG;
```

Members

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

The **DMUS_MIDI_PMSG** structure contains data for a standard MIDI message such as a control change or pitch bend.

Syntax

```
typedef struct DMUS_MIDI_PMSGG {
   DMUS_PMSG_PART
   BYTE bStatus;
   BYTE bByte1;
   BYTE bByte2;
   BYTE bPad[1];
} DMUS_MIDI_PMSG;
```

Members

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

- IDirectMusicPerformance8::SendPMsg
- <u>MIDI Messages</u>

DMUS_NOTE_PMSG

The **DMUS_NOTE_PMSG** structure contains message data for a MIDI note.

Syntax

ty	/pede1	f struct DMUS_NOTE_PMSG	{
DMUS_PMSG_PART			
	MUSIC	C_TIME mtDuration;	
	WORD	wMusicValue;	
	WORD	wMeasure;	
	short	t nOffset;	
	BYTE	bBeat;	
	BYTE	bGrid;	
	BYTE	bVelocity;	
	BYTE	bFlags;	
	BYTE	bTimeRange;	
	BYTE	bDurRange;	
	BYTE	<u> </u>	
	BYTE	bPlayModeFlags;	
	BYTE	bSubChordLevel;	
	BYTE	bMidiValue;	
	char	cTranspose;	
ſ	DMIIC	NATE DMCC.	

} DMUS_NOTE_PMSG;

Members

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.

Value Description

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 sho never be above 3. The first position in the chor	
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 <u>velocity</u>.

bFlags

See **<u>DMUS_NOTEF_FLAGS</u>**.

bTimeRange

Range by which to randomize the time at which the note plays.

bDurRange

Range by which to randomize the duration of the note.

bVelRange

Range by which to randomize the velocity of the note.

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 **<u>DMUS_SUBCHORD</u>**.

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.

The values in **bTimeRange** and **bDurRange** have a logarithmic relationship to actual time. A value of 255 specifies that the time can be randomized by up to 825 music time ticks in either direction.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>IDirectMusicPerformance8::SendPMsg</u>
- Music Values and MIDI Notes

DMUS_NOTIFICATION_PMSG

The **DMUS_NOTIFICATION_PMSG** structure contains message data for a notification.

Syntax

```
typedef struct DMUS_NOTIFICATION_PMSG {
   DMUS_PMSG_PART
   GUID guidNotificationType;
   DWORD dwNotificationOption;
   DWORD dwField1;
   DWORD dwField2;
} DMUS_NOTIFICATION_PMSG;
```

Members

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

guidNotificationType

Identifier of the notification type. The following types are defined.

Value	Description
GUID_NOTIFICATION_CHORD	Chord change.
GUID_NOTIFICATION_COMMAND	Command event.
GUID_NOTIFICATION_MEASUREANDBEAT	ГMeasure and beat event.
GUID_NOTIFICATION_PERFORMANCE	Performance event, further defined in dwNotificationOption .
GUID_NOTIFICATION_RECOMPOSE	A track has been recomposed.
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.

Value	Description
	The segment was stopped
DMUS_NOTIFICATION_SEGABORT	prematurely, or was removed
	from the primary segment queue.
DMUS_NOTIFICATION_SEGALMOSTEND	The segment has reached the end
DW05_NOTIFICATION_SEGALW05TENL	minus the <u>Prepare Time</u> .
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.

Value	Description
DMUS_NOTIFICATION_GROOVE	<u>Groove level</u> change.
DMUS NOTIFICATION EMBELLISHMENT	Embellishment command (intro,
DMUS_NOTIFICATION_EMBELLISHMENT	fill, break, or end).

If the notification type is GUID_NOTIFICATION_PERFORMANCE, this member can contain one of the following values.

Value	Description	
	The currently playing primary segment has reached the end	
DMUS_NOTIFICATION_MUSICALMOSTEND 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

- <u>IDirectMusicPerformance8::SendPMsg</u>
- Notification and Event Handling

DMUS_PATCH_PMSG

The **DMUS_PATCH_PMSG** structure contains message data for a MIDI program change.

Syntax

```
typedef struct DMUS_PATCH_PMSG {
   DMUS_PMSG_PART
   BYTE byInstrument;
   BYTE byMSB;
   BYTE byLSB;
   BYTE byPad[1];
} DMUS_PATCH_PMSG;
```

Members

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

byInstrument

Patch number of the instrument.

byMSB

Most significant byte of <u>bank select</u>.

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

- <u>DMUS_MIDI_PMSG</u>
- IDirectMusicPerformance8::SendPMsg
- <u>MIDI Messages</u>

DMUS_PMSG

The **DMUS_PMSG** structure 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.

Syntax

```
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
DWORD
              dwVoiceID;
             dwGroupID;
  IUnknown*
              punkUser;
} DMUS_PMSG;
```

Members

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. It must contain DMUS_PMSGF_REFTIME or DMUS_PMSGF_MUSICTIME.

dwPChannel

<u>Performance channel</u>. The port, channel group, and MIDI channel can be derived from this value by using the <u>IDirectMusicPerformance8::PChannelInfo</u> method. Set this value to zero 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.

Value	Description
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 zero if the message is not being sent by a track.

pTool

Address of the tool interface. Can be set by using **<u>IDirectMusicGraph8::StampPMsg</u>**, 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 **<u>IDirectMusicGraph8::StampPMsg</u>**, 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 zero.

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.

Value	Description
DMUS PMSGT STOP	Sending a message of this type stops the performance
DW05_PW3G1_310P	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.
	1

Requirements

Header: Declared in dmusici.h.

See Also

• <u>IDirectMusicPerformance8::SendPMsg</u>

DMUS_SYSEX_PMSG

The **DMUS_SYSEX_PMSG** structure contains data for a MIDI systemexclusive message.

Syntax

```
typedef struct DMUS_SYSEX_PMSG {
   DMUS_PMSG_PART
   DWORD dwLen;
   BYTE abData[1];
} DMUS_SYSEX_PMSG;
```

Members

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

- <u>DMUS_MIDI_PMSG</u>
- <u>DMUS_PATCH_PMSG</u>
- IDirectMusicPerformance8::SendPMsg
- <u>MIDI Messages</u>

DMUS_TEMPO_PMSG

The **DMUS_TEMPO_PMSG** structure contains data for a message that controls the performance's tempo.

Syntax

typedef struct DMUS_TEMPO_PMSG {
 DMUS_PMSG_PART
 double dblTempo;
} DMUS_TEMPO_PMSG;

Members

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

• <u>IDirectMusicPerformance8::SendPMsg</u>

DMUS_TIMESIG_PMSG

The **DMUS_TIMESIG_PMSG** structure contains data for a message that controls the time signature of the performance.

Syntax

```
typedef struct _DMUS_TIMESIG_PMSG {
   DMUS_PMSG_PART
   BYTE bBeatsPerMeasure;
   BYTE bBeat;
   WORD wGridsPerBeat;
} DMUS_TIMESIG_PMSG;
```

Members

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 <u>DMUS_SEGF_FLAGS</u>).

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 <u>Disabling and Enabling Track</u> <u>Parameters</u>.

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. For more informaton, see the **DMUS_NOTIFICATION_PMSG** structure.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DMUS_TIMESIGNATURE</u>
- IDirectMusicPerformance8::SendPMsg

DMUS_TRANSPOSE_PMSG

The **DMUS_TRANSPOSE_PMSG** structure contains message data for a transposition.

Syntax

```
typedef struct _DMUS_TRANSPOSE_PMSG {
  DMUS_PMSG_PART
  short nTranspose;
  WORD wMergeIndex;
} DMUS_TRANSPOSE_PMSG;
```

Members

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

The **DMUS_WAVE_PMSG** structure contains message data for a wave sound.

Syntax

```
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;
;
```

Members

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 one of the following values.

Value	Description
DMUS_WAVEF_IGNORELOOPS	Wave is not invalidated when a segment loop point is reached.
DMUS_WAVEF_NOINVALIDATE	Wave is not invalidated.
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.

Structure	Description
DMUS_AUDIOPARAMS	Describes required resources for the defa synthesizer and buffers of a performance
DMUS_BAND_PARAM	Used to set and retrieve band parameters
DMUS_BUFFERDESC	Describes a buffer for the IDirectMusic8::CreateMusicBuffer method.
DMUS_CHORD_KEY	Describes a chord in the IDirectMusicPerformance8::MIDITol and IDirectMusicPerformance8::MusicTo methods.
DMUS_CHORD_PARAM	Used to set and retrieve chord parameter Equivalent to DMUS_CHORD_KEY .
DMUS_CLOCKINFO8	Contains information about a clock enumerated by using the IDirectMusic8::EnumMasterClock m
DMUS_COMMAND_PARAM	Used to set and retrieve command track parameters.
DMUS_COMMAND_PARAM_2	Used to set and retrieve command track parameters with additional timing information.
DMUS_EVENTHEADER	Describes an event in a port buffer.
DMUS_MUTE_PARAM	Used to set and retrieve mute track parameters.
DMUS_NOTERANGE	Specifies a range of notes that an instrun must respond to.
DMUS_OBJECTDESC	Describe a loadable object.
DMUS PLAY MARKER PARAM	Contains information about a play mark

DMUS_PORTCAPS	Contains information about an enumerat port.
DMUS_PORTPARAMS8	Specifies parameters for the opening of a DirectMusic port.
DMUS_RHYTHM_PARAM	Contains parameters for chord rhythm.
DMUS_SCRIPT_ERRORINFO	Contains information about a script error
DMUS_SUBCHORD	Describes a subchord.
DMUS_SYNTHSTATS8	Describes the status of a synthesizer.
DMUS_TEMPO_PARAM	Used to set and retrieve tempo track parameters.
DMUS_TIMESIGNATURE	Contains information about a time signa in a style, style track, or time signature t
DMUS_VALID_START_PARAM	Used to retrieve valid start times from a marker track.
DMUS_VARIATIONS_PARAM	Contains information about variations associated with channels.
DMUS_VERSION	Contains version information for an obje
DMUS_WAVES_REVERB_PARAMS	Contains information about reverberatio effects on a DirectX 7.0 synthesizer.
KSPROPERTY	Used by the IKsControl::KsProperty method to identify a property and operat

Special categories of structures are contained in the following sections:

- <u>DirectMusic Messages</u>
- <u>DirectMusic File Structures</u>
- DLS Structures

DMUS_AUDIOPARAMS

The **DMUS_AUDIOPARAMS** structure describes required resources for the default synthesizer and buffers of a performance. It is passed to the **IDirectMusicPerformance8::InitAudio** method to request desired features and to receive information about what requests were granted.

Syntax

```
typedef struct _DMUS_AUDIOPARAMS {
  DWORD dwSize;
  BOOL fInitNow;
  DWORD dwValidData;
  DWORD dwFeatures;
  DWORD dwVoices;
  DWORD dwSampleRate;
  CLSID clsidDefaultSynth;
} DMUS_AUDIOPARAMS;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_AUDIOPARAMS)** 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.

Flag	Description
	The dwFeatures member

DMUS_AUDIOPARAMS_FEATURES	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.
The clsidDefaultSynth member contains or has received data. If DMUS_AUDIOPARAMS_DEFAULTSYNTH this flag is not set, the Microsoft software synthesizer is the default synthesizer.	

dwFeatures

Flags that specify required capabilities. The following values are defined.

Value	Description
	3-D buffers. This flag is not
DMUS_AUDIOF_3D	implemented. Buffers in 3-D audiopaths
	always have 3-D capabilities.
DMUS_AUDIOF_ALL	All features.
DMUS_AUDIOF_BUFFERS	Multiple buffers.
DMUS_AUDIOF_DMOS	Additional DMOs. This flag is not
	implemented.
DMUS_AUDIOF_ENVIRON	Environmental modeling. This flag is
	not implemented.
DMUS_AUDIOF_EAX	Support for Environmental Audio
	Extensions (EAX). This flag is not
	implemented.
DMUS_AUDIOF_STREAMING	Support for streaming waveforms.

dwVoices

Number of voices. The default value is 64.

dwSampleRate

Sample rate of the 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 dmusici.h.

See Also

• <u>DirectMusic Structures</u>

DMUS_BAND_PARAM

The **DMUS_BAND_PARAM** structure is 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 <u>GUID_BandParam</u>.

Syntax

```
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. When this structure is retrieved by a **GetParam** call, a reference to the band object is added. The application is responsible for releasing this reference.

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 dmusici.h.

See Also

- <u>DirectMusic Structures</u>
- IDirectMusicPerformance8::GetParam
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicSegment8::SetParam</u>
- <u>IDirectMusicTrack8::GetParamEx</u>
- <u>IDirectMusicTrack8::SetParamEx</u>
- <u>Performance Parameters</u>

DMUS_BUFFERDESC

The **DMUS_BUFFERDESC** structure is used to describe a buffer for the **IDirectMusic8::CreateMusicBuffer** method.

Syntax

```
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 to **sizeof(DMUS_BUFFERDESC)** 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

- <u>DirectMusic Structures</u>
- **DMUS_EVENTHEADER**
- <u>IDirectMusicBuffer8::PackStructured</u>
- IDirectMusicBuffer8::PackUnstructured

DMUS_CHORD_KEY

The **DMUS_CHORD_KEY** structure is used to describe a chord in the **IDirectMusicPerformance8::MIDIToMusic** and **IDirectMusicPerformance8::MusicToMIDI** methods.

Syntax

typedef struct _DMUS_CHORD_KEY { WCHAR wszName[16]; WORD wMeasure; BYTE bBeat; bSubChordCount; BYTE DMUS_SUBCHORD SubChordList[DMUS_MAXSUBCHORD]; DWORD dwScale; BYTE bKey; BYTE bFlags; } DMUS_CHORD_KEY;

Members

wszName

Name of the chord as specified in the Chordmap Designer component of DirectMusic Producer; for example, 2CM.

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. Each of the lower 24 bits represents a note in a two-octave scale, where position 0 is the root note of the scale.

bKey

Key underlying the entire chord, where 0 is C, 1 is C# or Bb, and so on.

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 <u>pattern</u> 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

- <u>DirectMusic Structures</u>
- IDirectMusicPerformance8::GetParam
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- IDirectMusicSegment8::SetParam
- IDirectMusicTrack8::GetParamEx
- <u>IDirectMusicTrack8::SetParamEx</u>

• <u>Performance Parameters</u>

DMUS_CHORD_PARAM

The **DMUS_CHORD_PARAM** structure is 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 <u>GUID_ChordParam</u>.

Syntax

typedef DMUS_CHORD_KEY DMUS_CHORD_PARAM;

Members

See **DMUS_CHORD_KEY**.

Requirements

Header: Declared in dmusicf.h.

See Also

- <u>DirectMusic Structures</u>
- IDirectMusicPerformance8::GetParam
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicSegment8::SetParam</u>
- <u>IDirectMusicTrack8::GetParamEx</u>
- IDirectMusicTrack8::SetParamEx
- <u>Performance Parameters</u>

DMUS_CLOCKINF08

The **DMUS_CLOCKINFO8** structure contains information about a clock enumerated by using the **IDirectMusic8::EnumMasterClock** method.

Syntax

typedef DMUS_CLOCKINF08 DMUS_CLOCKINF0; typedef DMUS_CLOCKINF0 *LPDMUS_CLOCKINF0;

Members

dwSize

Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_CLOCKINFO8)** 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. Can be 0 or DMUS_CLOCKF_GLOBAL.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>DirectMusic Structures</u>

DMUS_COMMAND_PARAM

The **DMUS_COMMAND_PARAM** structure is 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 <u>GUID_CommandParam</u>.

Syntax

typedef struct {
 BYTE bCommand;
 BYTE bGrooveLevel;
 BYTE bGrooveRange;
 BYTE bRepeatMode;
} DMUS_COMMAND_PARAM;

Members

bCommand

Command type. See **DMUS_COMMANDT_TYPES**.

bGrooveLevel

<u>Groove level</u> 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 <u>patterns</u> are selected from among multiple matching patterns. See <u>DMUS_PATTERNT_TYPES</u>.

Requirements

Header: Declared in dmusici.h.

See Also

See Also

- <u>DirectMusic Structures</u>
- IDirectMusicPerformance8::GetParam
- IDirectMusicPerformance8::SetParam
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicSegment8::SetParam</u>
- <u>IDirectMusicTrack8::GetParamEx</u>
- <u>IDirectMusicTrack8::SetParamEx</u>

DMUS_COMMAND_PARAM_2

The **DMUS_COMMAND_PARAM_2** structure is 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 <u>GUID_CommandParam2</u>.

Syntax

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

<u>Groove level</u> 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 <u>patterns</u> are selected for repetition. See <u>DMUS_PATTERNT_TYPES</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Structures</u>
- IDirectMusicPerformance8::GetParam
- <u>IDirectMusicPerformance8::SetParam</u>
- IDirectMusicSegment8::GetParam
- IDirectMusicSegment8::SetParam
- IDirectMusicTrack8::GetParamEx
- <u>IDirectMusicTrack8::SetParamEx</u>

DMUS_EVENTHEADER

The **DMUS_EVENTHEADER** structure precedes and describes an event in a port buffer.

Syntax

```
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

- <u>DirectMusic Structures</u>
- <u>IDirectMusicBuffer8::GetNextEvent</u>
- IDirectMusicBuffer8::PackStructured
- IDirectMusicBuffer8::PackUnstructured

DMUS_MUTE_PARAM

The **DMUS_MUTE_PARAM** structure is 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 <u>GUID_MuteParam</u>.

Syntax

```
typedef struct _DMUS_MUTE_PARAM {
  DWORD dwPChannel;
  DWORD dwPChannelMap;
  BOOL fMute;
} DMUS MUTE PARAM;
```

Members

dwPChannel

<u>Performance channel</u> 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 want all the notes on channel 3 to play on channel 9 instead, set **dwPChannel** to 3 and **dwPChannelMap** to 9 before passing the structure to one of the set methods. If you want to mute the notes on PChannel 8, set **dwPChannel** to 8 and **dwPChannelMap** to 0xFFFFFFF.

Requirements

Header: Declared in dmusicf.h.

See Also

- DirectMusic Structures
- <u>IDirectMusicPerformance8::GetParam</u>
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicSegment8::SetParam</u>
- IDirectMusicTrack8::GetParamEx
- IDirectMusicTrack8::SetParamEx

DMUS_NOTERANGE

The **DMUS_NOTERANGE** structure 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.

Syntax

```
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.

See Also

• DirectMusic Structures

DMUS_OBJECTDESC

The **DMUS_OBJECTDESC** structure is used to describe a loadable 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.

Syntax

```
typedef struct _DMUS_OBJECTDESC {
                dwSize;
  DWORD
  DWORD
                dwValidData;
                guidObject;
  GUID
  GUID
                quidClass;
                ftDate;
  FILETIME
  DMUS_VERSION vVersion;
  WCHAR
                wszName[DMUS_MAX_NAME];
                wszCategory[DMUS_MAX_CATEGORY];
 WCHAR
 WCHAR
                wszFileName[DMUS_MAX_FILENAME];
  LONGLONG
                llMemLength;
  LPBYTE
                pbMemData;
                pStream
  IStream*
} 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:

Flag	Description
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. See <u>DirectMusic Component GUIDs</u>.

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

Pointer to data in memory. Do not use this value except when loading from a resource contained in the executable file.

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 contain 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 clone of the stream object, and this clone 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**. The actual data of the stream is not copied, so the application should not change or delete the data.

Requirements

Header: Declared in dmusici.h.

See Also

• DirectMusic Structures

DMUS_PLAY_MARKER_PARAM

The **DMUS_PLAY_MARKER_PARAM** structure contains information about a play marker.

Syntax

```
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

- DirectMusic Structures
- <u>GUID_Play_Marker</u>

DMUS_PORTCAPS

The **DMUS_PORTCAPS** structure 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.

Syntax

```
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:

Flag	Description
DMUS_PC_AUDIOPATH	Multiple outputs can be connected to DirectSound for audiopaths.
DMUS_PC_DIRECTSOUND	The port supports streaming waveform data to DirectSound.

DMUS_PC_DLS	The port supports <u>DLS</u> 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 waveforms 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:

Value	Description
DMUS_PC_INPUTCLASS	Input port.
DMUS_PC_OUTPUTCLASS Output port.	

dwType

Type of this port. The following types are defined:

Value	Description
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:

Flag	Description
DMUS_EFFECT_NONE	No effects are supported.
DMUS_EFFECT_REVERB	The port supports reverb.
DMUS_EFFECT_CHORUS	The port supports chorus.
DMUS_EFFECT_DELAY	The port supports delay.

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.

See Also

• <u>DirectMusic Structures</u>

DMUS_PORTPARAMS8

The **DMUS_PORTPARAMS8** structure 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**.

Syntax

```
typedef struct _DMUS_PORTPARAMS8 {
   DWORD dwSize;
   DWORD dwValidParams;
   DWORD dwVoices;
   DWORD dwChannelGroups;
   DWORD dwAudioChannels;
   DWORD dwSampleRate;
   DWORD dwEffectFlags;
   DWORD dwEffectFlags;
   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 wanted. The following values are defined:

Flag	Description
DMUS_EFFECT_NONE	No effects are wanted.
DMUS_EFFECT_REVERB	Reverb is wanted.
DMUS_EFFECT_CHORUS	Chorus is wanted.
DMUS_EFFECT_DELAY	Delay is wanted.

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.

Flag	Description
DMUS_PORT_FEATURE_AUDIOPATH	Supports an audiopath connection to DirectSound buffers.
DMUS_PORT_FEATURE_STREAMING	Supports streaming waveforms through the synthesizer.

Requirements

Header: Declared in dmusicc.h.

See Also

- <u>DirectMusic Structures</u>
- <u>DMUS_PORTCAPS</u>

DMUS_RHYTHM_PARAM

The **DMUS_RHYTHM_PARAM** structure is used as the *pParam* parameter in calls to the various get-parameter methods when the track is a chord track and *rguidType* is <u>GUID_RhythmParam</u>.

Syntax

```
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

- DirectMusic Structures
- <u>IDirectMusicPerformance8::GetParam</u>
- IDirectMusicPerformance8::SetParam
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicSegment8::SetParam</u>
- <u>IDirectMusicTrack8::GetParamEx</u>
- <u>IDirectMusicTrack8::SetParamEx</u>

DMUS_SCRIPT_ERRORINFO

The **DMUS_SCRIPT_ERRORINFO** structure contains information about a script error.

Syntax

```
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_ERRORINF0;
```

Members

dwSize

Size of this structure, in bytes. This member must be initialized to **sizeof(DMUS_SCRIPT_ERRORINFO)** 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.

See Also

• <u>DirectMusic Structures</u>

DMUS_SUBCHORD

The **DMUS_SUBCHORD** structure is used in the **SubChordList** member of a **DMUS_CHORD_PARAM** structure.

Syntax

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 10000111111 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.

Remarks

Chords authored in DirectMusic Producer can have up to four component subchords, each assigned to a different level. Different parts in a pattern can be assigned to different chord levels, so that when the pattern plays, the transpositions within different parts may vary. In the DirectMusic API, complex chords can be added to a track by using the <u>GUID_ChordParam</u> parameter.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>DirectMusic Structures</u>

DMUS_SYNTHSTATS8

The DMUS_SYNTHSTATS8 structure is used by the

IDirectMusicPort8::GetRunningStats method to return the current running status of a synthesizer.

Syntax

typedef	<pre>struct DMUS_SYNTHSTATS {</pre>
DWORD	dwSize;
DWORD	dwValidStats;
DWORD	dwVoices;
DWORD	dwTotalCPU;
DWORD	dwCPUPerVoice;
DWORD	dwLostNotes;
DWORD	dwFreeMemory;
long	lPeakVolume;
DWORD	dwSynthMemUse;
} DMUS_S	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 <u>DLS</u> 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 waveform 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 dmusicc.h.

See Also

<u>DirectMusic Structures</u>

DMUS_TEMPO_PARAM

The **DMUS_TEMPO_PARAM** structure is 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 <u>GUID_TempoParam</u>.

Syntax

```
typedef struct _DMUS_TEMPO_PARAM {
   MUSIC_TIME mtTime;
   double dblTempo;
} DMUS_TEMPO_PARAM;
```

Members

mtTime

Time for which the tempo was retrieved. This is an offset from the time requested in the *mtTime* parameter of the **GetParam** or **GetParamEx** method. This member is ignored in calls to **SetParam**.

dblTempo

The tempo, in the range from DMUS_TEMPO_MIN through DMUS_TEMPO_MAX.

Requirements

Header: Declared in dmusicf.h.

See Also

- DirectMusic Structures
- <u>IDirectMusicPerformance8::GetParam</u>
- IDirectMusicPerformance8::SetParam
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicSegment8::SetParam</u>
- IDirectMusicTrack8::GetParamEx
- <u>IDirectMusicTrack8::SetParamEx</u>

DMUS_TIMESIGNATURE

The **DMUS_TIMESIGNATURE** structure is 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.

Syntax

```
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 <u>DMUS_SEGF_FLAGS</u>).

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Structures</u>
- <u>DMUS_TIMESIG_PMSG</u>
- IDirectMusicPerformance8::GetParam
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicSegment8::SetParam</u>
- <u>IDirectMusicTrack8::GetParamEx</u>, <u>IDirectMusicTrack8::SetParamEx</u>

DMUS_VALID_START_PARAM

The **DMUS_VALID_START_PARAM** structure is used as the *pParam* parameter in calls to various get-parameter methods when *rguidType* is <u>GUID_Valid_Start_Time</u>.

Syntax

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

- DirectMusic Structures
- IDirectMusicPerformance8::GetParam
- <u>IDirectMusicSegment8::GetParam</u>
- IDirectMusicTrack8::GetParamEx

DMUS_VARIATIONS_PARAM

The **DMUS_VARIATIONS_PARAM** structure contains information about variations associated with channels. It is used when retrieving the <u>GUID_Variations</u> parameter.

Syntax

```
typedef struct _DMUS_VARIATIONS_PARAM {
  DWORD dwPChannelsUsed;
  DWORD* padwPChannels;
  DWORD* padwVariations;
} DMUS_VARIATIONS_PARAM;
```

Members

dwPChannelsUsed

The number of performance channels in use.

padwPChannels

Address of an array of performance channels in use.

padwVariations

Address of an array of variations in effect for each channel, where each bit set represents a variation.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>DirectMusic Structures</u>

DMUS_VERSION

The **DMUS_VERSION** structure contains version information for an object described in the **DMUS_OBJECTDESC** structure.

Syntax

```
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.

See Also

• DirectMusic Structures

DMUS_WAVES_REVERB_PARAMS

The **DMUS_WAVES_REVERB_PARAMS** structure contains information about reverberation effects in the Microsoft software synthesizer provided with DirectX 7.0.

Syntax

```
typedef struct _DMUS_WAVES_REVERB_PARAMS {
  float fInGain;
  float fReverbMix;
  float fReverbTime;
  float fHighFreqRTRatio;
} DMUS_WAVES_REVERB_PARAMS;
```

Members

fInGain

Input gain, in decibels. The default value is 0.

fReverbMix

Reverb mix, in decibels. A value of 0 means 100 percent wet reverb (no direct signal). Negative values give a drier 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 a very bright reverb is 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 <u>DMO</u>.

Requirements

Header: Declared in dmusicc.h.

See Also

• <u>DirectMusic Structures</u>

KSPROPERTY

The **KSPROPERTY** structure is used by the **IKsControl::KsProperty** method to identify a property and operation.

KSPROPERTY is defined as a **KSIDENTIFIER** structure, which is declared as follows.

Syntax

```
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:

Value	Description
GUID_DMUS_PROP_DLS1	Item 0 is a Boolean indicating whet or not this port supports downloadi DLS level 1 samples.
GUID_DMUS_PROP_DLS2	Item 0 is a Boolean indicating whet or not this port supports downloadi 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 proper used to set or retrieve the current st of the effects.
GUID_DMUS_PROP_GM_Hardware	Item 0 is a Boolean indicating whet or not this port supports GM in hardware.
GUID_DMUS_PROP_GS_Capable	Item 0 is a Boolean indicating whet or not this port supports the minimu Requirements for Roland GS extensions.
GUID_DMUS_PROP_GS_Hardware	Item 0 is a Boolean indicating whet or not this port supports Roland GS extensions in hardware.
GUID_DMUS_PROP_INSTRUMENT2	Item 0 is a Boolean indicating whet or not this port supports downloadi: samples using the <u>DMUS_ARTICULATION2</u> struct
GUID_DMUS_PROP_LegacyCaps	Item 0 is the MIDIINCAPS or MIDIOUTCAPS structure that describes the underlying Windows multimedia device implementing th port. A MIDIINCAPS structure is returned if dwClass is DMUS_PC_INPUTCLASS in this port's capabilities structure. Otherw a MIDIOUTCAPS structure is returned.
GUID_DMUS_PROP_MemorySize	Item 0 is the number of bytes of sar RAM free on this device.
GUID_DMUS_PROP_SampleMemorySize	Item 0 is the number of bytes of sar RAM, both free and used, available this device.
GUID_DMUS_PROP_SamplePlaybackRate	Item 0 is the synthesizer's sample ra The DLS level 2 file format suppor conditional chunks to determine whether a region or <u>articulation</u> sho be downloaded. This allows author create optional waveforms intended

	different sample rates. Setting a san rate of 96 kHz or greater may cause reverberation effects to fail.
GUID_DMUS_PROP_SynthSink_DSOUND	Item 0 is a Boolean indicating whet or not this port supports DirectSour
GUID_DMUS_PROP_SynthSink_WAVE	Item 0 is a Boolean indicating whet or not this port supports waveform output using the waveOut function
GUID_DMUS_PROP_Volume	Item 1 (DMUS_ITEM_Volume) is LONG in the range from DMUS_VOLUME_MAX through DMUS_VOLUME_MIN. This is th signed value, in hundredths of a dec which is added to the gain of all vol after all DLS articulation has been performed. By default, when a port added to the performance, this prop is set to the master volume. For mas volume, see Setting and Retrieving Global Parameters.
GUID_DMUS_PROP_WavesReverb	Item 0 is a DMUS_WAVES_REVERB_PAR structure containing reverb paramet
GUID_DMUS_PROP_WriteLatency	Item 0 is a user-defined value in milliseconds (in the range 0 to 100) is added to the latency of a user-mo synthesizer that sends its output to DirectSound. The default value is 5 The latency is the delay between w the synthesizer writes data to a buff and when the data is sent to the dev Increasing the latency can solve sor sound breakup problems. The prope must be reset each time the port is activated.
	Item 0 is the period, in milliseconds the range 2 to 100), at which the processing thread runs when the

GUID_DMUS_PROP_WritePeriod	synthesizer is a user-mode synthesi that sends its output to DirectSounc The default value is 10. If your application requires the absolute lov latency possible, you can set this property to values smaller than the default value of 10 to process data 1 often, but performance will suffer because of frequent context switchi To reduce CPU consumption at the expense of latency, set this property more than 10. The value is the same all port instances that use the standa DirectSound sink. The property mu set each time the port is activated.
GUID_DMUS_PROP_XG_Capable	Item 0 is a Boolean indicating whet or not this port supports the minimu Requirements for Yamaha XG extensions.
GUID_DMUS_PROP_XG_Hardware	Item 0 is a Boolean indicating whet or not this port supports Yamaha X extensions in hardware.

Id

Item within the property set.

Flags

One of the following flags to specify the operation:

Flag	Description
KSPROPERTY_TYPE_GET	Retrieve the given property item's value.
KSPROPERTY_TYPE_SET	Set the given property item's value.
	Ascertain the type of support available for the property set. The data returned by IKsControl::KsProperty in

KSPROPERTY_TYPE_BASICSUPPORT *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

- <u>DirectMusic Structures</u>
- <u>Property Sets for DirectMusic Ports</u>

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 <u>Low-Level DLS</u>.

For more information on DLS data formats, see the specification from the MIDI Manufacturers Association.

The following structures are included in this section.

Structure	Description
DMUS_ARTICPARAMS	Describes parameters for a DLS <u>articulation</u> chunk.
DMUS_ARTICULATION	Describes a DLS level 1 articulation chunk.
DMUS_ARTICULATION2	Describes a DLS level 1 or level 2 articulation chunk.
DMUS_COPYRIGHT	Describes an optional copyright chunk in DLS data
DMUS_DOWNLOADINFO	Used as a header for DLS data to be downloaded to a port.
DMUS_EXTENSIONCHUNK	Describes a DLS extension chunk.
DMUS_INSTRUMENT	Contains an instrument definition in a DLS download chunk.
DMUS_LFOPARAMS	Defines the low-frequency oscillator for a DLS articulation chunk
DMUS_MSCPARAMS	Defines the pan for a DLS articulation chunk.
DMUS_OFFSETTABLE	Used in the header of DLS instrument data being downloaded to a port.

DMUS_PEGPARAMS	Defines the pitch envelope for a DLS level 1 articulation chunk.
DMUS_REGION	Defines a region for a DLS download.
DMUS_VEGPARAMS	Defines a volume envelope for a DLS level 1 articulation chunk.
DMUS_WAVE	Defines a wave chunk for a DLS download.
DMUS_WAVEARTDL	Contains information for downloading waveform articulation.
DMUS_WAVEDATA	Contains a data chunk for a DLS waveform download.
DMUS_WAVEDL	ontains information about waveform data downloaded to the synthesizer

DMUS_ARTICPARAMS

The **DMUS_ARTICPARAMS** structure describes parameters for a <u>DLS</u> 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 <u>DMUS_ARTICULATION</u> chunk.

Syntax

```
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.

See Also

• <u>DLS Structures</u>

DMUS_ARTICULATION

The **DMUS_ARTICULATION** structure 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**.

Syntax

```
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

- <u>DLS Structures</u>
 <u>DMUS_ARTICULATION2</u>

DMUS_ARTICULATION2

The **DMUS_ARTICULATION2** structure 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**.

Syntax

```
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

- DLS Structures
- **<u>DMUS_ARTICULATION</u>**

DMUS_COPYRIGHT

The **DMUS_COPYRIGHT** structure describes an optional copyright chunk in <u>DLS</u> data.

Syntax

```
typedef struct {
   ULONG cbSize;
   BYTE byCopyright[ ];
} DMUS_COPYRIGHT;
```

Members

cbSize

Size of data.

byCopyright[]

Copyright data.

Requirements

Header: Declared in dmdls.h.

See Also

DLS Structures

DMUS_DOWNLOADINFO

The **DMUS_DOWNLOADINFO** structure is used as a header for <u>DLS</u> 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.

Syntax

```
typedef struct _DMUS_DOWNLOADINFO {
  DWORD dwDLType;
  DWORD dwDLId;
  DWORD dwNumOffsetTableEntries;
  DWORD cbSize;
} DMUS_DOWNLOADINF0;
```

Members

dwDLType

Type of data being downloaded. The following types are defined:

Description
Instrument definition, starting
with the
DMUS_INSTRUMENT
structure.
Instrument definition supporting
DLS level 2 articulation, starting
2 with the
DMUS_INSTRUMENT
structure.
PCM waveform 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 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

- DLS Structures
- Low-Level DLS

DMUS_EXTENSIONCHUNK

The **DMUS_EXTENSIONCHUNK** structure describes a <u>DLS</u> extension chunk. All extensions to the DLS file format that are unknown to DirectMusic are downloaded in this variable-size chunk.

Syntax

```
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.

See Also

• DLS Structures

DMUS_INSTRUMENT

The **DMUS_INSTRUMENT** structure contains an instrument definition in a <u>DLS</u> download chunk.

Syntax

```
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:

Flag	Description
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
	Гinstruments. 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.

See Also

DLS Structures

DMUS_LFOPARAMS

The **DMUS_LFOPARAMS** structure defines the low-frequency oscillator for a <u>DLS articulation</u> chunk. It is used in the **DMUS_ARTICPARAMS** structure.

Syntax

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

- DLS Structures
- **DMUS_ARTICPARAMS**

DMUS_MSCPARAMS

The **DMUS_MSCPARAMS** structure defines the pan for a <u>DLS</u> <u>articulation</u> chunk. This structure is used in the <u>DMUS_ARTICPARAMS</u> structure.

Syntax

```
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

- DLS Structures
- **DMUS_ARTICPARAMS**

DMUS_OFFSETTABLE

The **DMUS_OFFSETTABLE** structure is used in the header of <u>DLS</u> instrument data being downloaded to a port.

Syntax

```
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

- DLS Structures
- Low-Level DLS

DMUS_PEGPARAMS

The **DMUS_PEGPARAMS** structure defines the pitch envelope for a <u>DLS</u> level 1 <u>articulation</u> chunk. It is used in the <u>DMUS_ARTICPARAMS</u> structure.

Syntax

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

- DLS Structures
- **DMUS_ARTICPARAMS**

DMUS_REGION

The **DMUS_REGION** structure defines a region for a <u>DLS</u> download. One or more regions can be embedded in an instrument buffer and referenced by the instrument header chunk, <u>DMUS_INSTRUMENT</u>.

Syntax

```
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:

Flag	Description
	If a second note-on for the same
	note is received by the synthesis
	engine, the second note is played,
F_RGN_OPTION_SELFNONEXCLUSIVE	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 waveform. The **ulTableIndex** member of the **WAVELINK** structure contains the download identifier of the associated 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 waveform. If the **cSampleLoops** member is 1, the following **WLOOP** structure

carries the loop start and end points.

WLOOP[]

Structure describing a loop. The **WLOOP** type is declared in Dls1.h.

Requirements

Header: Declared in dmdls.h.

See Also

• <u>DLS Structures</u>

DMUS_VEGPARAMS

The **DMUS_VEGPARAMS** structure defines a volume envelope for a <u>DLS</u> level 1 <u>articulation</u> chunk.

Syntax

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

• **<u>DMUS_ARTICPARAMS</u>**

See Also

• <u>DLS Structures</u>

DMUS_WAVE

The **DMUS_WAVE** structure defines a wave chunk for a <u>DLS</u> download.

Syntax

```
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 waveform.

ulCopyrightIdx

Index, in the **DMUS_OFFSETTABLE** structure, of copyright chunks. If 0, no copyright information is associated with the waveform.

ulWaveDataIdx

Index, in the **DMUS_OFFSETTABLE** structure, of waveform data. See **DMUS_WAVEDATA**.

WaveformatEx

WAVEFORMATEX structure that specifies the WAV format of the chunk.

Requirements

Header: Declared in dmdls.h.

See Also

• DLS Structures

DMUS_WAVEARTDL

The **DMUS_WAVEARTDL** structure contains information for downloading waveform <u>articulation</u>.

Syntax

```
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.

See Also

• <u>DLS Structures</u>

DMUS_WAVEDATA

The **DMUS_WAVEDATA** structure contains a data chunk for a <u>DLS</u> waveform download. The nature of the data is defined by the **WaveformatEx** member of the <u>DMUS_WAVE</u> structure.

Syntax

```
typedef struct {
   ULONG cbSize;
   BYTE byData[ ];
} DMUS_WAVEDATA;
```

Members

cbSize

Size of data.

byData[]

PCM waveform data.

Requirements

Header: Declared in dmdls.h.

See Also

DLS Structures

DMUS_WAVEDL

ontains information about waveform data downloaded to the synthesizer.

Syntax

typedef struct _DMUS_WAVEDL {
 ULONG cbWaveData;
} DMUS_WAVEDL, *LPDMUS_WAVEDL;

Members

cbWaveData

Number of bytes of data.

Requirements

Header: Declared in dmdls.h.

See Also

DLS Structures

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:

- <u>About RIFF</u>
- <u>RIFF Notation</u>

The following topics describe how DirectMusic data is organized in RIFF chunks:

- <u>Common Chunks</u>
- Audiopath Form
- Band Form
- <u>Chordmap Form</u>
- <u>Container Form</u>
- DirectSound Buffer Configuration Form
- Effects Form
- <u>Reference List</u>
- <u>Script Form</u>
- Segment Form
- <u>Style Form</u>
- <u>Tool Form</u>
- <u>Toolgraph Form</u>
- Track Form
- <u>Wave Header Chunk</u>

The following section is a reference to structures used to contain data in RIFF chunks:

• <u>DirectMusic File Structures</u>

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 <u>styles</u> 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></element>	File element labeled "element", or of
	type element .
[<element>]</element>	Optional file element.
<element></element>	One or more copies of the specified
	element.
[<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 <chehck> 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_I0_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(<<u>DMUS IO VERSION</u>>)

<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

<u>Reference List</u>

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 <u>Common Chunks</u>.

<DMTG-form>

See <u>Toolgraph Form</u>.

<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( <<u>DMUS IO PORTCONFIG HEADER</u>> )
<pprh-ck> -> pprh( <<u>DMUS PORTPARAMS8</u>> )
```

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 <u>DirectSound Buffer Configuration Form</u>. 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 <u>performance channels</u> to buffers.

This list consists of an array of chunks, each of which describes one assignment of channels to buffers.

```
<pchh-ck> -> pchh(
    <<u>DMUS IO PCHANNELTOBUFFER HEADER</u>>
    <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, cpcsl-list>.

```
© 2004 Microsoft Corporation. All rights reserved.
```

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 <u>Common Chunks</u>.

<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( <<u>DMUS IO INSTRUMENT</u>> )
```

The instrument list can also contain <DMRF-list>, which in this case is a reference to a <u>DLS</u> file. See <u>Reference List</u>.

```
© 2004 Microsoft Corporation. All rights reserved.
```

Chordmap Form

The following notation shows the organization of the top-level chunk, or form, of a <u>chordmap</u> file:

```
RIFF( 'DMPR'
   <perh-ck> // Chordmap 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 <u>Common Chunks</u>.

<chdt-ck>

```
<chdt-ck> -> chdt(
  <WORD> // Size of DMUS_I0_CHORDMAP_SUBCHORD.
  <<u>DMUS IO CHORDMAP_SUBCHORD</u>>...
)
```

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( <<u>DMUS IO CHORDENTRY</u>> )
```

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.
    <<u>DMUS IO NEXTCHORD</u>>...
)
```

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( <<u>DMUS IO CHORDMAP SIGNPOST</u>> )
```

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 <u>style</u> forms. It can be contained within a segment or script file. It is organized as follows:

<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 <u>Common Chunks</u>.

<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 (<<u>DMUS IO CONTAINED OBJECT HEADER</u>>)

The header is normally followed by object data of the type specified in <cobhck>. This can be in the form of a RIFF chunk such as a <u>Segment Form</u> or a <u>Style</u> <u>Form</u>. If it is a DMRF-list, it is a reference to the object. For more information on DMRF, see <u>Reference List</u>.

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 <u>Common Chunks</u>.

<dsbd-ck>

The DirectSound buffer descriptor is organized as follows:

```
<dsbd-ck> -> 'dsbd' (
<<u>DSOUND IO DSBUFFERDESC</u>>
)
```

<bsid-ck>

The bus identifiers are stored in the following chunk.

<bsid-ck> -> 'bsid' (<<u>DSOUND IO DSBUSID</u>>)

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'(<<u>DSOUND IO 3D</u>>)

<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 <u>DMO</u> effect.

```
RIFF ( 'DSFX'
<fxhr-ck>
[<data-ck>]
)
```

<fxhr-ck>

The header chunk describes the effect.

```
<fxhr-ck> -> fxhr ( <<u>DSOUND IO DXDMO HEADER</u>> )
```

<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 (<<u>DSOUND IO DXDMO DATA</u>>)

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 <u>DLS</u> 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( <<u>DMUS IO REFERENCE</u>> )
```

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_I0_SCRIPT_HEADER> )
```

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see <u>Common Chunks</u>.

<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 <u>Container Form</u>.

<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 <u>Reference List</u> chunk pointing to a text file containing the script code. The **guidClassID** member of the <u>DMUS_IO_REFERENCE</u> 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 <u>Container Form</u>.

```
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( <<u>DMUS IO SEGMENT HEADER</u>> )
```

<guid-ck>, <vers-ck>, <UNFO-list>

See Common Chunks.

<DMCN-form>

See <u>Container Form</u>.

<trkl-list>

This is the track list. Each track is encapsulated in a <u>Track Form</u>.

```
<trkl-list> -> LIST( 'trkl'
<DMTK-form>...
)
```

<DMTG-form>

See **Toolgraph Form**.

<DMAP-form>

See <u>Audiopath Form</u>.

Style Form

The following notation shows the organization of the top-level chunk of a <u>style</u> file. This form can also be contained within a <u>Container Form</u>.

<styh-ck>

This chunk contains the basic header information for a style.

```
<styh-ck> -> styh( <<u>DMUS IO STYLE</u>> )
```

<guid-ck>,<UNFO-list>, <vers-ck>

For information on these three chunks, see <u>Common Chunks</u>.

<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 <u>Common Chunks</u>.)

```
<prth-ck> -> prth( <<u>DMUS IO STYLEPART</u>> )
<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.
  < <u>DMUS IO STYLEMARKER</u> >...
  )
<rsln-ck> -> rsln(
  < DWORD > // Size of DMUS_IO_STYLERESOLUTION.
  < DMUS IO STYLERESOLUTION >...
  )
<anpn-ck> -> anpn(
  < DWORD > Size of DMUS IO STYLE ANTICIPATION.
  < <u>DMUS_IO_STYLE_ANTICIPATION</u> >...
  )
```

<pttn-list>

Each pattern is described in a chunk with the following format.

The first chunk of the pattern list is a header:

```
<ptnh-ck> -> ptnh( <<u>DMUS IO PATTERN</u>> )
```

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**.

For the optional UNFO list, see <u>Common Chunks</u>.

The next chunk of the pattern list describes the motif settings:

<mtfs-ck> -> mtfs(<<u>DMUS IO MOTIFSETTINGS</u>>)

For the <DMBD-form> chunk of the pattern list, see <u>Band Form</u>.

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(<<u>DMUS IO PARTREF</u>>)

<DMBD-form>

The next chunk in the style form is a **Band Form**.

<prrf-list>

The final chunk contains an array of <u>chordmap</u> references:

```
<prrf-list> -> LIST( 'prrf'
<DMRF-list>...
)
```

For more information on <DMRF-list>, see <u>Reference List</u>.

Tool Form

The tool form contains information about tools. Tools can be embedded in a <u>Toolgraph Form</u> 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(<<u>DMUS IO TOOL HEADER</u>>)

<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 <u>Segment Form</u> or <u>Container Form</u>.

```
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 <u>Common Chunks</u>.

<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 <u>Tool Form</u>.

Track Form

The track form contains information about a single track. It can be embedded in a <u>Segment Form</u> or stored in its own file.

<trkh-ck>

The first chunk contains the basic header information for a track.

```
<trkh-ck> -> trkh( <<u>DMUS IO TRACK HEADER</u>> )
```

<trkx-ck>

This optional chunk contains flags for the track.

<trkx-ck> -> trkx(<<u>DMUS IO TRACK EXTRAS HEADER</u>>)

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see <u>Common Chunks</u>.

<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 <u>DMUS_IO_TRACK_HEADER</u> structure. The following standard track chunks are defined:

- Band Track Form
- <u>Chord Track List</u>
- <u>Chordmap Track List</u>
- Command Track Chunk

- Lyrics Track List
- <u>Marker Track List</u>
- <u>Mute Track Chunk</u>
- Parameter Control Track List
- <u>Pattern Track Form</u>
- <u>Script Track List</u>
- <u>Segment Trigger Track List</u>
- <u>Sequence Track Chunk</u>
- Signpost Track Chunk
- <u>Style Track List</u>
- Sysex Track Chunk
- <u>Tempo Track Chunk</u>
- <u>Time Signature Track List</u>
- 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 <u>Track Form</u>. 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(<<u>DMUS IO BAND TRACK HEADER</u>>)

<guid-ck>, <vers-ck>, <UNFO-list>

For information on these three chunks, see <u>Common Chunks</u>.

<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> -> (<<u>DMUS IO BAND ITEM HEADER</u>>)

<bd2h-ck> -> (<<u>DMUS IO BAND ITEM HEADER2</u>>)

The header is followed by a <u>Band Form</u> containing information about the instruments in the band.

Chord Track List

The chord track list contains chord data for a <u>Track Form</u>. It is organized as follows:

```
<cord-list> -> LIST( 'cord'
  <crdh-ck> // Header
  <crdb-ck>... // Chord body chunks
)
```

<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_I0_CHORD
  <DMUS IO CHORD>
  <DWORD> // Number of subchords
  <DWORD> // Size of DMUS_I0_SUBCHORD
  <DMUS IO SUBCHORD>...
)
```

Chordmap Track List

The chordmap track list contains data for a <u>Track Form</u>. 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 <u>Reference List</u>.

Command Track Chunk

The command track chunk contains data for a <u>Track Form</u>. It is organized as follows:

```
<cmnd-ck> -> cmnd(
  <DWORD> //Size of DMUS_IO_COMMAND
  <<u>DMUS IO COMMAND</u>>...
)
```

Lyrics Track List

The lyrics track list contains data for a <u>Track Form</u>.

```
<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(<<u>DMUS IO LYRICSTRACK EVENTHEADER</u>>)

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 <u>Track Form</u>. 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
<<u>DMUS IO VALID_START</u>>...
)
```

The second element is an array of chunks defining valid play points.

```
<play-ck> -> play(
DWORD // size of DMUS_IO_PLAYMARKER
<<u>DMUS IO PLAYMARKER</u>>...
)
```

Mute Track Chunk

The mute track chunk contains data for a <u>Track Form</u>. It is organized as follows:

```
<mute-ck> -> mute(
  <DWORD> //Size of DMUS_IO_MUTE
  <<u>DMUS IO MUTE</u>>...
)
```

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(<<u>DMUS IO PARAMCONTROLTRACK OBJECTHEADER</u>>)

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(<<u>DMUS IO PARAMCONTROLTRACK PARAMHEADER</u>>)

The array of curves is described in the following chunk.

```
c
```

Pattern Track Form

The pattern track form can be a top-level form but is also found as data for a <u>Track Form</u>. It is organized as follows:

```
RIFF( 'DMPT'
<styh-ck> // Style header chunk
<pttn-list> // The pattern
)
```

For information on <styh-ck>, see <u>Style Form</u>.

The <pattn-list> used in the pattern track is not the same as the one used in the Style Form. The pattern track version also contains part lists.

Script Track List

The script track list contains data for a <u>Track Form</u>.

```
<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 <u>Reference List</u>.

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 <u>Track Form</u>. 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' ( <<u>DMUS IO SEGMENT TRACK HEADER</u>> )
```

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> -> ( <<u>DMUS IO SEGMENT ITEM HEADER</u>> )
```

This is followed by a reference to a segment file or a <u>style</u> file. It is a reference to a style if the DMUS_SEGMENTTRACKF_MOTIF flag is present in the item header. For more information, see <u>Reference List</u>.

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 <u>Track Form</u>. 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 <u>curve</u> items:

```
<evtl-ck> -> evtl(
   <DWORD> // Size of DMUS_I0_SEQ_ITEM
   <DMUS IO SEQ_ITEM>...
   )
<curl-ck> -> curl(
   <DWORD> // Size of DMUS_I0_CURVE_ITEM
   <DMUS_I0_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 <u>Track Form</u>. It is organized as follows:

```
<sgnp-list> -> sgnp(
  <DWORD> // Size of DMUS_I0_SIGNPOST.
  <<u>DMUS IO SIGNPOST</u>>...
)
```

Style Track List

The style track list contains data for a <u>Track Form</u>. 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 <u>Reference List</u>.

Sysex Track Chunk

The sysex track chunk contains data for a <u>Track Form</u>. It is an array of system exclusive message items, each consisting of a <u>DMUS_IO_SYSEX_ITEM</u> structure followed by the number of bytes specified in the **dwSysExLength** member.

```
<syex-ck> -> 'syex' (
{
  <<u>DMUS IO SYSEX ITEM</u>>
  <BYTE>... // Data
}...
)
```

Tempo Track Chunk

The tempo track chunk contains data for a <u>Track Form</u>. It is organized as follows:

```
<tetr-ck> -> tetr(
  <DWORD> // Size of DMUS_IO_TEMPO_ITEM.
  <<u>DMUS IO TEMPO ITEM</u>>...
)
```

Time Signature Track List

The time signature track list contains data for a <u>Track Form</u>. 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_I0_TIMESIGNATURE_ITEM.
  <<u>DMUS IO TIMESIGNATURE ITEM</u>>...
)
```

Wave Track List

The following notation shows the organization of a chunk containing data for a wave track:

<wath-ck>

This chunk contains header information for a wave. It is followed by an array of lists describing wave parts:

```
<wath-ck> -> wath( <<u>DMUS IO WAVE TRACK HEADER</u>> )
```

<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' (<<u>DMUS IO WAVE PART HEADER</u>>)

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:

)

The wave description begins with a header chunk:

<waih-ck> -> (<<u>DMUS IO WAVE ITEM HEADER</u>>)

For more information on <DMRF-list>, see <u>Reference List</u>.

Wave Header Chunk

The wave header chunk is a special chunk added to WAV files for DirectMusic. It specifies streaming capabilities.

<wavh-ck> -> (<<u>DMUS IO WAVE HEADER</u>>)

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:

Structure	Description
DMUS_IO_BAND_ITEM_HEADER	Contains inform change. Superse DMUS_IO_B A
DMUS_IO_BAND_ITEM_HEADER2	Contains inforn change.
DMUS_IO_BAND_TRACK_HEADER	Contains inform behavior of a ba
DMUS_IO_BUFFER_ATTRIBUTES_HEADER	Describes attrib buffer.
DMUS_IO_CHORD	Contains inforn change.
DMUS_IO_CHORDENTRY	Contains inforn entry.
DMUS_IO_CHORDMAP	Contains inform
DMUS_IO_CHORDMAP_SIGNPOST	Contains inforn chord in a chord
DMUS_IO_CHORDMAP_SUBCHORD	Contains inform
DMUS_IO_COMMAND	Contains inforn event.
DMUS_IO_CONTAINED_OBJECT_HEADER	Used before eac Form.
DMUS_IO_CONTAINER_HEADER	Used in the Cor
	Contains inform

DMUS_IO_CURVE_ITEM	in a track.
DMUS_IO_INSTRUMENT	Contains inform instrument.
DMUS_IO_LYRICSTRACK_EVENTHEADER	Used in a Lyric:
DMUS_IO_MOTIFSETTINGS	Contains inform
DMUS_IO_MUTE	Contains inform on a channel.
DMUS_IO_NEXTCHORD	Contains inform chord in a chord
DMUS_IO_PARAMCONTROLTRACK_CURVEINFO	Used in a Paran
DMUS_IO_PARAMCONTROLTRACK_OBJECTHEADER	Used in a Paran
DMUS_IO_PARAMCONTROLTRACK_PARAMHEADER	Used in a Paran
DMUS_IO_PARTREF	Contains inform
	reference.
DMUS_IO_PATTERN	Contains inform
DMUS_IO_PCHANNELTOBUFFER_HEADER	Defines a range and the buffers
DMUS_IO_PLAYMARKER	Contains inform marker, which i segment where permitted to sta
DMUS_IO_PORTCONFIG_HEADER	Contains inform configuration.
DMUS_IO_REFERENCE	Contains inform to another objec another file.
DMUS_IO_SCRIPT_HEADER	Used in the Scri
DMUS_IO_SCRIPTTRACK_EVENTHEADER	Used in a Script
DMUS_IO_SEGMENT_HEADER	Contains inform
DMUS_IO_SEGMENT_ITEM_HEADER	Contains inform referenced in a List.
DMUS_IO_SEGMENT_TRACK_HEADER	Contains inforn Trigger Track L
DMUS_IO_SEQ_ITEM	Contains inform data in a sequer

DMUS_IO_SIGNPOST	Contains inform a signpost track signpost chords
DMUS_IO_STYLE	Contains inform signature and te
DMUS_IO_STYLE_ANTICIPATION	Describes a reso
DMUS_IO_STYLECURVE	Contains inform style.
DMUS_IO_STYLEMARKER	Contains inform style.
DMUS_IO_STYLENOTE	Contains inform style.
DMUS_IO_STYLEPART	Contains inform part.
DMUS_IO_STYLERESOLUTION	Describes a styl
DMUS_IO_SUBCHORD	Contains inform
DMUS_IO_SYSEX_ITEM	Contains inform exclusive MIDI
DMUS_IO_TEMPO_ITEM	Contains inform change in a trac
DMUS_IO_TIMESIG	Contains inform signature of a se
DMUS_IO_TIMESIGNATURE_ITEM	Contains inform signature chang
DMUS_IO_TOOL_HEADER	Contains inform
DMUS_IO_TRACK_EXTRAS_HEADER	Used in the Tra
DMUS_IO_TRACK_HEADER	Contains inform
DMUS_IO_VALID_START	Contains inform point in a segme rhythm.
DMUS_IO_VERSION	Contains the ve
DMUS_IO_WAVE_HEADER	Describes streai wave.
DMUS_IO_WAVE_ITEM_HEADER	Contains data fo Wave Track Lis
DMUS_IO_WAVE_PART_HEADER	Contains data fo

DMUS_IO_WAVE_TRACK_HEADER	Contains data fo Track List.
DSOUND_IO_3D	Contains 3-D pa DirectSound bu
DSOUND_IO_DSBUFFERDESC	Describes a Dir
DSOUND_IO_DSBUSID	Contains bus id
DSOUND_IO_DXDMO_DATA	Contains data fo
DSOUND_IO_DXDMO_HEADER	Contains heade

See Also

• <u>DirectMusic File Format</u>

DMUS_IO_BAND_ITEM_HEADER

The **DMUS_IO_BAND_ITEM_HEADER** structure contains information about a band change. Used in the <u>Band Track Form</u> of older files. It has been superseded by **DMUS_IO_BAND_ITEM_HEADER2**.

Syntax

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.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_BAND_ITEM_HEADER2

The **DMUS_IO_BAND_ITEM_HEADER2** structure contains information about a band change. Used in the <u>Band Track Form</u>.

Syntax

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.

IBandTimePhysical

Precise time when band change will take effect. Should be close to logical time.

Requirements

Header: Declared in dmusicf.h.

See Also

- <u>DirectMusic File Structures</u>
- <u>Segment Timing</u>

DMUS_IO_BAND_TRACK_HEADE

The **DMUS_IO_BAND_TRACK_HEADER** structure contains information about the default behavior of a band track. Used in the <u>Band Track Form</u>.

Syntax

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 <u>Using Bands</u>.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_BUFFER_ATTRIBUTES_

The **DMUS_IO_BUFFER_ATTRIBUTES_HEADER** structure describes attributes of a DirectSound buffer. Used in the <u>Audiopath Form</u>.

Syntax

```
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.

Value

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.

Value	Description
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.

See Also

• DirectMusic File Structures

DMUS_IO_CHORD

The **DMUS_IO_CHORD** structure contains information about a chord change. Used in the <u>Chord Track List</u>.

Syntax

```
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 as shown.

Value	Description
	The chord is silent. See the Remarks for

Requirements

Header: Declared in dmusicf.h.

See Also

• DirectMusic File Structures

DMUS_IO_CHORDENTRY

The **DMUS_IO_CHORDENTRY** structure contains information about a chord entry. Used in the <u>Chordmap Form</u>.

Syntax

```
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.

See Also

• DirectMusic File Structures

DMUS_IO_CHORDMAP

The **DMUS_IO_CHORDMAP** structure contains information about a <u>chordmap</u>. Used in the <u>Chordmap Form</u>.

Syntax

```
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.

Value	Description
DMUS_CHORDMAPF_VERSION8	The chordmap was created for DirectX 8.0 or later.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_CHORDMAP_SIGNPOST

The **DMUS_IO_CHORDMAP_SIGNPOST** structure contains information about a signpost chord in a <u>chordmap</u>. Used in the <u>Chordmap Form</u>.

Syntax

```
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 <u>templates</u>. 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.

Value

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

See Also

- <u>DirectMusic File Structures</u>
- <u>DMUS_IO_SIGNPOST</u>

DMUS_IO_CHORDMAP_SUBCHOF

The **DMUS_IO_CHORDMAP_SUBCHORD** structure contains information about a subchord. Used in the <u>Chordmap Form</u>.

Syntax

```
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 <u>style</u> 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

- <u>DirectMusic File Structures</u>
- <u>DMUS_SUBCHORD</u>

DMUS_IO_COMMAND

The **DMUS_IO_COMMAND** structure contains information about a command event. Used in the <u>Command Track Chunk</u>.

Syntax

```
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 <u>DMUS_COMMANDT_TYPES</u>.

bGrooveLevel

<u>Groove level</u>, 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 <u>DMUS_PATTERNT_TYPES</u>.

Requirements

Header: Declared in dmusicf.h.

See Also

• DirectMusic File Structures

DMUS_IO_CONTAINED_OBJECT_]

The **DMUS_IO_CONTAINED_OBJECT_HEADER** structure is used before each object in a <u>Container Form</u>.

Syntax

```
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.

Value	Description
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.

See Also

• DirectMusic File Structures

DMUS_IO_CONTAINER_HEADER

The **DMUS_IO_CONTAINER_HEADER** structure is used in the <u>Container</u> <u>Form</u>.

Syntax

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.

Value	Description
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.

See Also

• DirectMusic File Structures

DMUS_IO_CURVE_ITEM

The **DMUS_IO_CURVE_ITEM** structure contains information about a curve event in a track. Used in the <u>Sequence Track Chunk</u>.

Syntax

```
typedef struct _DMUS_IO_CURVE_ITEM {
  MUSIC_TIME mtStart;
 MUSIC_TIME mtDuration;
 MUSIC_TIME mtResetDuration;
 DWORD dwPChannel;
         nOffset;
  short
  short
         nStartValue;
  short nEndValue;
  short
         nResetValue;
 BYTE
         bType;
 BYTE
BYTE
         bCurveShape;
         bCCData;
 BYTE
         bFlags;
 WORD
         wParamType;
         wMergeIndex
 WORD
} 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 as shown.

Туре	Description
DMUS_CURVET_CCCURVE	Continuous controller curve (MIDI Control Change channel voice message; status byte &HB <i>n</i> , where <i>n</i> is the channel number).
DMUS_CURVET_MATCURVE	Monophonic aftertouch curve (MIDI Channel Pressure channel voice message; status byte &HDn).
DMUS_CURVET_PATCURVE	Polyphonic aftertouch curve (MIDI Poly Key Pressure channel voice message, status byte &HDn).
DMUS_CURVET_PBCURVE	Pitch-bend curve (MIDI Pitch Bend channel voice message; status byte &HE <i>n</i>).
DMUS_CURVET_NRPNCURVE	<u>NRPN</u> curve.
DMUS_CURVET_RPNCURVE	<u>RPN</u> curve.

bCurveShape

Shape of curve. The following shapes are defined as shown.

Shape	Description
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 <u>RPN</u> and <u>NRPN</u> 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

• <u>DirectMusic File Structures</u>

- <u>Curves</u>
 <u>DMUS_IO_SEQ_ITEM</u>

DMUS_IO_INSTRUMENT

The **DMUS_IO_INSTRUMENT** structure contains information about an instrument. Used in the **Band Form**.

Syntax

```
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

<u>Performance channel</u> that the instrument plays on.

dwFlags

Control flags. The following values are defined as shown.

Value	Description
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 Dank-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.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_LYRICSTRACK_EVENT

The **DMUS_IO_LYRICSTRACK_EVENTHEADER** structure is used in a Lyrics Track List.

Syntax

```
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 members of the **DMUS_PMSGF_FLAGS** enumeration shown in the following table.

Value	Description
	Message should be processed
DMUS_PMSGF_TOOL_IMMEDIATE	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

- <u>DirectMusic File Structures</u>
- <u>Segment Timing</u>

DMUS_IO_MOTIFSETTINGS

The **DMUS_IO_MOTIFSETTINGS** structure contains information about a motif. Used in the <u>Style Form</u>.

Syntax

```
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 **<u>DMUS_TIME_RESOLVE_FLAGS</u>**.

Requirements

Header: Declared in dmusicf.h.

See Also

- <u>DirectMusic File Structures</u>
- IDirectMusicSegment8::SetLoopPoints

DMUS_IO_MUTE

The **DMUS_IO_MUTE** structure contains information about a mute event on a channel. Used in the <u>Mute Track Chunk</u>.

Syntax

```
typedef struct _DMUS_IO_MUTE {
   MUSIC_TIME mtTime;
   DWORD dwPChannel;
   DWORD dwPChannelMap;
} DMUS_IO_MUTE;
```

Members

mtTime

Time of the event.

dwPChannel

<u>Performance channel</u> 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

- <u>DirectMusic File Structures</u>
- <u>DMUS_MUTE_PARAM</u>

DMUS_IO_NEXTCHORD

The **DMUS_IO_NEXTCHORD** structure contains information about the next chord in a chord graph. Used in the <u>Chordmap Form</u>.

Syntax

```
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.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_PARAMCONTROLTRAC

The **DMUS_IO_PARAMCONTROLTRACK_CURVEINFO** structure is used in a **Parameter Control Track List**.

Syntax

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.

See Also

• DirectMusic File Structures

DMUS_IO_PARAMCONTROLTRAC

The **DMUS_IO_PARAMCONTROLTRACK_OBJECTHEADER** structure is used in a <u>Parameter Control Track List</u>.

Syntax

```
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

<u>Performance channel</u>, 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.

See Also

• DirectMusic File Structures

DMUS_IO_PARAMCONTROLTRAC

The **DMUS_IO_PARAMCONTROLTRACK_PARAMHEADER** structure is used in a <u>Parameter Control Track List</u>.

Syntax

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.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_PARTREF

The **DMUS_IO_PARTREF** structure contains information about a part reference. Used in the <u>Style Form</u>.

Syntax

```
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.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_PATTERN

The **DMUS_IO_PATTERN** structure contains information about a pattern. Used in the <u>Style Form</u>.

Syntax

```
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 or more of the constants from the **DMUS_EMBELLISHT_TYPES** enumeration, or a value defined by the content provider, such as a custom embellishment number assigned in DirectMusic Producer.

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 value shown in the following table.

Description
Variation settings in the state data
L of a pattern-based track persist in
the track after it stops playing.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_PCHANNELTOBUFFER_

The **DMUS_IO_PCHANNELTOBUFFER_HEADER** structure defines a range of <u>performance channels</u> and the buffers they connect to. Used in the port configuration list of an <u>Audiopath Form</u>.

Syntax

```
typedef struct _DMUS_IO_PCHANNELTOBUFFER_HEADER {
   DWORD dwPChannelBase;
   DWORD dwPChannelCount;
   DWORD dwBufferCount;
   DWORD dwFlags;
}
```

} DMUS_I0_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.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_PLAYMARKER

The **DMUS_IO_PLAYMARKER** structure 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 <u>Marker Track List</u>.

Syntax

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.

See Also

• DirectMusic File Structures

DMUS_IO_PORTCONFIG_HEADEI

The **DMUS_IO_PORTCONFIG_HEADER** structure contains information about a port configuration. It is used in the <u>Audiopath Form</u>.

Syntax

```
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.

Value	Description
DMUS_PORTCONFIGF_DRUMSON	10 Drums are on channel 10.
DMUS_PORTCONFIGF_USEDEFAU	LT Use the default port.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_REFERENCE

The **DMUS_IO_REFERENCE** structure contains information about a reference to another object that might be stored in another file. Used in the <u>Reference List</u> chunk.

Syntax

```
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.

See Also

• DirectMusic File Structures

DMUS_IO_SCRIPT_HEADER

The **DMUS_IO_SCRIPT_HEADER** structure is used in the <u>Script Form</u>.

Syntax

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 values shown in the following table.

Value	Description
DMUS_SCRIPTIOF_LOAD_ALL_CONTENT	All content in the script
	If DMUS_SCRIPTIOF
	the bands from all the s
	are downloaded when t
	a segment's bands are d
	the segment.
DMUS_SCRIPTIOF_DOWNLOAD_ALL_SEGMENTS	5
	If
	DMUS_SCRIPTIOF_E
	is not set, the script mu
	the segment's bands.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_SCRIPTTRACK_EVENT

The **DMUS_IO_SCRIPTTRACK_EVENTHEADER** structure is used in a <u>Script Track List</u>.

Syntax

```
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 values shown in the following table.

Value	Description	
	Fire the event in advance of the time	
	stamp, at prepare time. This is the	
DMUS_IO_SCRIPTTRACKF_PREPARE	MUS_IO_SCRIPTTRACKF_PREPARE 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

• <u>Segment Timing</u>

See Also

• DirectMusic File Structures

DMUS_IO_SEGMENT_HEADER

The **DMUS_IO_SEGMENT_HEADER** structure contains information about a segment. Used in the <u>Segment Form</u>.

Syntax

```
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;
   REFERENCE_TIME rtLoopStart;
   REFERENCE_TIME rtLoopEnd;
   REFERENCE_TIME rtLoopEnd;
   REFERENCE_TIME rtPlayStart;
}
DWUS TO SECMENT HEADEP;
```

} 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 0 or one or more of the following flags.

Value	Description
DMUS_SEGIOF_REFLENGTH	The value in rtLength overrides mtLength .
DMUS_SEGIOF_CLOCKTIME	The segment is played in clock time.

dwReserved

Reserved.

rtLoopStart

Loop start in clock time.

rtLoopEnd

Loop end in clock time.

rtPlayStart

Start point in clock time.

Requirements

Header: Declared in dmusicf.h.

See Also

- <u>DMUS_IO_MOTIFSETTINGS</u>
- <u>IDirectMusicSegment8::SetLoopPoints</u>

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_SEGMENT_ITEM_HEAI

The **DMUS_IO_SEGMENT_ITEM_HEADER** structure contains information about a segment referenced in the <u>Segment Trigger Track List</u>.

Syntax

```
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.

ITimePhysical

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.

Value	Description
	The DMRF chunk is a link to a <u>style</u> ,
DMUS_SEGMENTTRACKF_MOTIF	and the 'snam' chunk is the name of a
	motif within the style.

Requirements

Header: Declared in dmusicf.h.

See Also

• DirectMusic File Structures

DMUS_IO_SEGMENT_TRACK_HE

The **DMUS_IO_SEGMENT_TRACK_HEADER** structure contains information about a <u>Segment Trigger Track List</u>.

Syntax

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.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_SEQ_ITEM

The **DMUS_IO_SEQ_ITEM** structure contains information about an item of data in a sequence track. Used in the <u>Sequence Track Chunk</u>.

Syntax

```
typedef struct _DMUS_IO_SEQ_ITEM {
   MUSIC_TIME mtTime;
   MUSIC_TIME mtDuration;
   DWORD dwPChannel;
   short nOffset
   BYTE bStatus;
   BYTE bByte1;
   BYTE bByte2;
}
DMUS_IO_SFO_ITEM:
```

} DMUS_I0_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

- <u>DirectMusic File Structures</u>
- <u>DMUS_IO_CURVE_ITEM</u>
- <u>MIDI Messages</u>

DMUS_IO_SIGNPOST

The **DMUS_IO_SIGNPOST** structure contains information about a signpost in a signpost track to associate it with signpost chords in a <u>chordmap</u>. Used in the <u>Signpost Track Chunk</u>.

Syntax

```
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 <u>templates</u>. 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 as shown.

Value

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

- DirectMusic File Structures
- <u>DMUS_IO_CHORDMAP_SIGNPOST</u>

DMUS_IO_STYLE

The **DMUS_IO_STYLE** structure contains information about the time signature and tempo of a <u>style</u>. Used in the <u>Style Form</u>.

Syntax

```
typedef struct _DMUS_I0_STYLE {
   DMUS_I0_TIMESIG timeSig;
   double dblTempo;
} DMUS_I0_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.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_STYLE_ANTICIPATION

The **DMUS_IO_STYLE_ANTICIPATION** structure describes a resolution anticipation. Used in the <u>Style Form</u>.

Syntax

```
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.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_STYLECURVE

The **DMUS_IO_STYLECURVE** structure contains information about a <u>curve</u> in a <u>style</u>. Used in the <u>Style Form</u>.

Syntax

```
typedef struct _DMUS_I0_STYLECURVE {
  MUSIC_TIME mtGridStart;
  DWORD dwVariation;
  MUSIC_TIME mtDuration;
  MUSIC_TIME mtResetDuration;
  short
          nTimeOffset;
  short
          nStartValue;
  short nEndValue;
short nResetValue;
 BYTE
BYTE
BYTE
BYTE
WORD
          bEventType;
          bCurveShape;
          bCCData;
          bFlags;
          wParamType;
  WORD
          wMergeIndex;
} DMUS_I0_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

<u>RPN</u> or <u>NRPN</u> 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

- <u>DirectMusic File Structures</u>
- <u>DMUS_CURVE_PMSG</u>
- <u>DMUS_IO_CURVE_ITEM</u>

DMUS_IO_STYLEMARKER

The **DMUS_IO_STYLEMARKER** structure contains information about a marker in a <u>style</u>. Used in the <u>Style Form</u>.

Syntax

```
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.

Value	Description
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.

See Also

• DirectMusic File Structures

DMUS_IO_STYLENOTE

The **DMUS IO STYLENOTE** structure contains information about a note in a style. Used in the Style Form.

Syntax

- typedef struct _DMUS_IO_STYLENOTE { MUSIC_TIME mtGridStart; DWORD dwVariation; MUSIC_TIME mtDuration; short nTimeOffset; WORD BYTE BYTE BYTE wMusicValue; bVelocity; bTimeRange; bDurRange; bVelRange; BYTE BYTE BYTE BYTE bInversionID; bPlayModeFlags; 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)</pre>
```

```
{
    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.

See Also

• DirectMusic File Structures

DMUS_IO_STYLEPART

The **DMUS_IO_STYLEPART** structure contains information about a musical part. Used in the <u>Style Form</u>.

Syntax

```
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:
```

```
} DMUS_I0_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 as shown.

Value	Description
DMUS_VARIATIONF_MAJOR	Seven positions in the scale for major chords.
DMUS_VARIATIONF_MINOR	Seven positions in the scale for mino

	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 sc 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	K 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 (dwVariationChoice DMUS_VARIATIONF_MODES_EX == DMUS_VARIATIONF_IMA25_MO the variations are in Interactive Music Architecture mode.
DMUS_VARIATIONF_DMUS_MODE	Mode mask. If dwVariationChoices contains this mask, the variations are DirectMusic mode. All variations authored in DirectMusic Producer us 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 values shown in the following table.

Value	Description
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.

See Also

• DirectMusic File Structures

DMUS_IO_STYLERESOLUTION

The **DMUS_IO_STYLERESOLUTION** structure describes a <u>style</u> resolution. Used in the <u>Style Form</u>.

Syntax

```
typedef struct _DMUS_I0_STYLERESOLUTION {
   DWORD dwVariation;
   WORD wMusicValue;
   BYTE bInversionID;
   BYTE bPlayModeFlags;
} DMUS_I0_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.

See Also

• DirectMusic File Structures

DMUS_IO_SUBCHORD

The **DMUS_IO_SUBCHORD** structure contains information about a subchord. Used in the <u>Chord Track List</u>.

Syntax

```
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 10000111111 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

- <u>DirectMusic File Structures</u>
- <u>DMUS_SUBCHORD</u>

DMUS_IO_SYSEX_ITEM

The **DMUS_IO_SYSEX_ITEM** structure contains information about a system exclusive MIDI message. Used in the <u>Sysex Track Chunk</u>.

Syntax

```
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

- DirectMusic File Structures
- <u>MIDI System Messages</u>

DMUS_IO_TEMPO_ITEM

The **DMUS_IO_TEMPO_ITEM** structure contains information about a tempo change in a track. Used in the <u>Tempo Track Chunk</u>.

Syntax

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.

See Also

• DirectMusic File Structures

DMUS_IO_TIMESIG

The **DMUS_IO_TIMESIG** structure contains information about the time signature of a segment. Used in the **DMUS_IO_STYLE**, **DMUS_IO_VERSION**, and **DMUS_IO_PATTERN** structures.

Syntax

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

- <u>DirectMusic File Structures</u>
- <u>DMUS_IO_TIMESIGNATURE_ITEM</u>

DMUS_IO_TIMESIGNATURE_ITEN

The **DMUS_IO_TIMESIGNATURE_ITEM** structure contains information about a time signature change. Used in the <u>Time Signature Track List</u>.

Syntax

```
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

- <u>DirectMusic File Structures</u>
- DMUS_IO_TIMESIG
 DMUS_TIMESIG_PMSG

DMUS_IO_TOOL_HEADER

The **DMUS_IO_TOOL_HEADER** structure contains information about a tool. Used in the <u>Tool Form</u>.

Syntax

```
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 <u>performance channels</u> for which the tool is valid.

Requirements

Header: Declared in dmusicf.h.

See Also

- <u>DirectMusic File Structures</u>
- IDirectMusicGraph8::InsertTool

DMUS_IO_TRACK_EXTRAS_HEAI

The **DMUS_IO_TRACK_EXTRAS_HEADER** structure is used in the <u>Track</u> <u>Form</u>.

Syntax

typedef struct _DMUS_IO_TRACK_EXTRAS_HEADER {
 DWORD dwFlags;

DWORD dwPriority;

} DMUS_IO_TRACK_EXTRAS_HEADER;

Members

dwFlags

Flags for control tracks. For possible values, see **IDirectMusicSegment8::SetTrackConfig**.

dwPriority

Priority for composition.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_TRACK_HEADER

The **DMUS_IO_TRACK_HEADER** structure contains information about a track. Used in the <u>Track Form</u>.

Syntax

```
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.

See Also

- <u>DirectMusic File Structures</u>
- IDirectMusicSegment8::GetTrackGroup
 IDirectMusicSegment8::InsertTrack
- <u>Track Form</u>

DMUS_IO_VALID_START

The **DMUS_IO_VALID_START** structure contains information about a valid start point in a segment that is to be cued to a rhythm. Used in the <u>Marker Track</u> <u>List</u>.

Syntax

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.

See Also

• DirectMusic File Structures

DMUS_IO_VERSION

The **DMUS_IO_VERSION** structure contains the version number of the data. Used in the version subchunk of various chunks. See <u>Common Chunks</u>.

Syntax

```
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.

See Also

• DirectMusic File Structures

DMUS_IO_WAVE_HEADER

The **DMUS_IO_WAVE_HEADER** structure describes streaming characteristics of a wave. It is used in the <u>Wave Header Chunk</u> of a WAV file.

Syntax

```
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 one or more of the following values.

Description
Preroll data is not downloaded with the
wave.
The wave is streamed.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic File Structures</u>

DMUS_IO_WAVE_ITEM_HEADER

The **DMUS_IO_WAVE_ITEM_HEADER** structure contains data for a wave sound in a <u>Wave Track List</u>.

Syntax

```
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
DWORD
DWORD
WORD
WORD
           dwLoopStart;
           dwLoopEnd;
           dwFlags;
           wVolumeRange;
           wPitchRange;
```

} 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 values in the following table.

Value	Description
DMUS_WAVEF_IGNORELOOPS	Wave is not invalidated when a segment
	loop point is reached.
DMUS_WAVEF_NOINVALIDATE	This wave is not to be invalidated.
DMUS_WAVEF_STREAMING	Wave is streaming.

wVolumeRange

Amount by which volume can be randomized, in hundredths of a decibel.

wPitchRange

Amount by which pitch can be randomized, in hundredths of a semitone.

Requirements

Header: Declared in dmusicf.h.

See Also

- <u>DirectMusic File Structures</u>
- <u>Segment Timing</u>

DMUS_IO_WAVE_PART_HEADER

The **DMUS_IO_WAVE_PART_HEADER** structure contains data for a <u>Wave</u> <u>Track List</u>.

Syntax

```
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

IVolume

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.

Requirements

Header: Declared in dmusicf.h.

See Also

• DirectMusic File Structures

DMUS_IO_WAVE_TRACK_HEADE

The **DMUS_IO_WAVE_TRACK_HEADER** structure contains data for a wave track in a <u>Wave Track List</u>.

Syntax

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 values shown in the following table.

Value	Description
DMUS_WAVETRACKF_SYNC_VAR	The track gets its variations from a pattern track. For more information, see <u>GUID_Variations</u> .
DMUS_WAVETRACKF_PERSIST_CONTROI	Variation control information persists from one playback instance to the next.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic File Structures</u>

DSOUND_IO_3D

The **DSOUND_IO_3D** structure contains 3-D parameters for a DirectSound buffer in a <u>DirectSound Buffer Configuration Form</u>.

Syntax

```
typedef struct _DSOUND_IO_3D {
  GUID guid3DAlgorithm;
  DS3DBUFFER ds3d;
} DSOUND_IO_3D;
```

Members

guid3DAlgorithm

Unique identifier of the 3-D algorithm to use.

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.

See Also

• DirectMusic File Structures

DSOUND_IO_DSBUFFERDESC

The **DSOUND_IO_DSBUFFERDESC** structure describes a DirectSound buffer. Used in the <u>DirectSound Buffer Configuration Form</u>.

Syntax

```
typedef struct _DSOUND_IO_DSBUFFERDESC {
   DWORD dwFlags;
   WORD nChannels;
   LONG lVolume;
   LONG lPan;
   DWORD dwReserved;
} DSOUND_IO_DSBUFFERDESC;
```

Members

dwFlags

Buffer creation flags.

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.

See Also

• DirectMusic File Structures

DSOUND_IO_DSBUSID

The **DSOUND_IO_DSBUSID** structure contains bus identifiers. Used in the <u>DirectSound Buffer Configuration Form</u>.

Syntax

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.

See Also

• DirectMusic File Structures

DSOUND_IO_DXDMO_DATA

The **DSOUND_IO_DXDMO_DATA** structure contains data for a <u>DMO</u>.

Syntax

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.

See Also

• <u>DirectMusic File Structures</u>

DSOUND_IO_DXDMO_HEADER

The **DSOUND_IO_DXDMO_HEADER** structure contains header information for a <u>DMO</u> chunk in an <u>Effects Form</u>.

Syntax

```
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.

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 <u>send</u> effect.

dwReserved

Reserved. Must be 0.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic File Structures</u>

Standard Track Parameters

This section describes the standard track parameters that can be set and retrieved by using the following methods.

- <u>IDirectMusicPerformance8::GetParam</u>
- <u>IDirectMusicPerformance8::GetParamEx</u>
- <u>IDirectMusicPerformance8::SetParam</u>
- <u>IDirectMusicSegment8::GetParam</u>
- <u>IDirectMusicSegment8::SetParam</u>
- <u>IDirectMusicTrack8::GetParam</u>
- <u>IDirectMusicTrack8::GetParamEx</u>
- <u>IDirectMusicTrack8::SetParam</u>
- <u>IDirectMusicTrack8::SetParamEx</u>

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 <u>Standard Track</u> <u>Types</u> . Although parameters are always associated with particular tracks, applications usually call the method on the segment or the performance and let DirectMusic find the
Data type	appropriate track. See <u>Identifying the Track</u> . Type of data pointed to by the <i>pParam</i> parameter of the method call by which the parameter is set or retrieved.
mtTime	Significance, if any, of the <i>mtTime</i> parameter of the method call by which the parameter is set or retrieved.

The standard parameter types are listed in the following table.

Parameter GUID	Description
GUID_BandParam	Sets or retrieves a band.
GUID_ChordParam	Sets or retrieves a chord change.

GUID_Clear_All_Bands	Clears all bands from the track.
GUID_CommandParam	Sets or retrieves a groove or embellishment command.
GUID_CommandParam2	Sets or retrieves a groove or embellishment command, with additional timing information.
GUID_CommandParamNext	Sets or retrieves a groove or embellishment command to follow all other commands in the command track.
GUID_ConnectToDLSCollection	Connects all bands in the track to a <u>DLS</u> collection.
GUID_Disable_Auto_Download	Disables automatic downloading of instruments and waves.
GUID_DisableTempo	Disables tempo messages.
GUID_DisableTimeSig	Disables time signature messages.
GUID_Download	Downloads instrument data or wave data to the performance.
GUID_DownloadToAudioPath	Downloads instrument data or wave data to an audiopath.
<u>GUID_Enable_Auto_Download</u>	Enables automatic downloading of instruments and waves.
GUID_EnableTempo	Enables tempo messages.
GUID_EnableTimeSig	Enables time signature messages.
GUID_IDirectMusicBand	Sets a band.
GUID_IDirectMusicChordMap	Sets or retrieves the <u>chordmap</u> .
GUID_IDirectMusicStyle	Sets or retrieves the <u>style</u> .
GUID_MuteParam	Sets or retrieves channel-mapping and muting information.
<u>GUID_Play_Marker</u>	Retrieves the next point in the currently playing segment at which a new segment can start.
<u>GUID_RhythmParam</u>	Retrieves the rhythm pattern for a sequence of chords stored in a measure in the track.
GUID_SeedVariations	Seeds the random number generator for variation selection.

<u>GUID_StandardMIDIFile</u>	Ensures that a standard MIDI file plays correctly.
GUID_TempoParam	Sets or retrieves the tempo.
GUID_TimeSignature	Retrieves the time signature.
GUID_Unload	Unloads instrument or wave data from the performance.
GUID_UnloadFromAudioPath	Unloads instrument or wave data from an audiopath.
<u>GUID_Valid_Start_Time</u>	Retrieves the next valid point within a segment at which it can start.
GUID_Variations	Retrieves the variations in effect across <u>performance channels</u> .

GUID_BandParam

Sets or retrieves a band.

Element	Description
Track type	Band
Data type (* <i>pParam</i>)	DMUS_BAND_PARAM
mtTime	Logical time at which to set the band, or the time for which to retrieve the band.

Remarks

When this parameter is retrieved by a **GetParam** call, a reference to the band object is created in **DMUS_BAND_PARAM.pBand**. The application is responsible for releasing this reference.

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

GUID_ChordParam

Sets or retrieves a chord change.

Element	Description
Track type	Chord
Data type (* <i>pParam</i>)	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 <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

GUID_Clear_All_Bands

Clears all bands from the track. This parameter can be set but not retrieved.

Element	Description	
Track type	Band	
Data type (* <i>pParam</i>)None		
mtTime	Not used	

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

GUID_CommandParam

Sets or retrieves a groove or embellishment command.

Element	Description
Track type	Command
Data type (* <i>pParam</i>)	DMUS_COMMAND_PARAM
	The time, in track time, at which to add the
mtTime	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 <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

GUID_CommandParam2

Sets or retrieves a groove or embellishment command.

Element	Description
Track type	Command
Data type (* <i>pParam</i>)	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 <u>Standard Track Parameters</u>.

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.

Element	Description	
Track type	Command	
Data type (* <i>pParam</i>) DMUS_COMMAND_PARAM_2		
mtTime	Not used	

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

GUID_ConnectToDLSCollection

Connects all bands in the track to a <u>DLS</u> collection. This parameter can be set but not retrieved.

Element	Description
Track type	Band
Data type (* <i>pParam</i>)	IDirectMusicCollection8 interface
	pointer
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>Playing a MIDI File with Custom Instruments</u>.

GUID_Disable_Auto_Download

Disables automatic downloading of instruments and waves. This parameter can be set but not retrieved.

Element	Description	
Track type	Band, wave	
Data type (<i>*pParam</i>)None		
mtTime	Not used	

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>GUID_Enable_Auto_Download</u>
- <u>Using Bands</u>

GUID_DisableTempo

Disables tempo messages. This parameter can be set but not retrieved.

Element	Description
Track type	Tempo
Data type (* <i>pParam</i>)None	
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- Disabling and Enabling Track Parameters
- <u>GUID_EnableTempo</u>

GUID_DisableTimeSig

Disables time signature messages. This parameter can be set but not retrieved.

Element	Description
Track type	Pattern, time signature, style, motif
Data type (* <i>pParam</i>)None	
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- Disabling and Enabling Track Parameters
- <u>GUID_EnableTimeSig</u>

GUID_Download

Downloads instrument data or wave data. This parameter can be set but not retrieved.

Element	Description
Track type	Band, wave
Data type (* <i>pParam</i>)	IDirectMusicPerformance8 interface pointer
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

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.

Description
Band, wave
IDirectMusicAudioPath8 interface
pointer
Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>IDirectMusicSegment8::Download</u>

GUID_Enable_Auto_Download

Enables automatic downloading of instruments and waves. This parameter can be set but not retrieved.

Element	Description
Track type	Band, wave
Data type (* <i>pParam</i>)None	
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>GUID_Disable_Auto_Download</u>
- <u>Using Bands</u>

GUID_EnableTempo

Enables tempo messages. This parameter can be set but not retrieved.

Element	Description	
Track type	Tempo	
Data type (* <i>pParam</i>)None		
mtTime	Not used	

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- Disabling and Enabling Track Parameters
- <u>GUID_DisableTempo</u>

GUID_EnableTimeSig

Enables time signature messages. This parameter can be set but not retrieved.

Element	Description
Track type	Pattern, time signature, style, motif
Data type (* <i>pParam</i>)None	
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- Disabling and Enabling Track Parameters
- <u>GUID_DisableTimeSig</u>

GUID_IDirectMusicBand

Sets a band. This parameter can be set but not retrieved.

Element	Description
Track type	Band
Data type (* <i>pParam</i>)	IDirectMusicBand8 interface pointer
mtTime	The time, in track time, at which to add the band to the track.

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

This parameter has been superseded by <u>GUID_BandParam</u>, which allows you to specify the physical time.

Requirements

Header: Declared in dmusici.h.

GUID_IDirectMusicChordMap

Sets or retrieves the <u>chordmap</u>.

Element	Description
Track type	Chordmap
Data type (* <i>pParam</i>)	IDirectMusicChordMap8 interface 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 <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

GUID_IDirectMusicStyle

Sets or retrieves the <u>style</u>.

Element	Description
Track type	Style
Data type (* <i>pParam</i>)	IDirectMusicStyle8 interface 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 <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

GUID_MuteParam

Sets or retrieves channel-mapping and muting information.

Element	Description
Track type	Mute.
Data type (* <i>pParam</i>)	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 <u>Standard Track Parameters</u>.

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.

Element	Description
Track type	Marker
Data type (* <i>pParam</i>)	DMUS_PLAY_MARKER_PARAM
mtTime	Track time at which to start seeking a marker

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>Segment Timing</u>

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.

Element	Description
Track type	Chord
Data type (* <i>pParam</i>)	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 <u>Standard Track Parameters</u>.

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.

Element	Description
Track type	Pattern, style, motif
Data type (*pParam)Long	
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

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.

Element	Description
Track type	Band
Data type (<i>*pParam</i>)None	
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Setting this parameter ensures that channels do not play silently if the file contains no patch changes. It also causes bank selects to be ignored. The parameter must be set before any instruments are downloaded.

Do not set this parameter if the file might contain DLS instrument data. Doing so causes the DSL instruments to be ignored and standard GM instruments to be used.

If you are writing an application to play MIDI content that might or might not contain DLS instruments, you should ascertain for each file whether or not the file contains a DLS chunk. The four-character code for this chunk is "DLS ", defined as FOURCC_DLS in Dls1.h. If the file contains this chunk, do not set the GUID_StandardMIDIFile parameter.

See Also

• <u>About RIFF</u>

Requirements

Header: Declared in dmusici.h.

^{© 2004} Microsoft Corporation. All rights reserved.

GUID_TempoParam

Sets or retrieves the tempo.

Element	Description
Track type	Тетро
Data type (* <i>pParam</i>)	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 <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

GUID_TimeSignature

Retrieves the time signature. This parameter can be retrieved but not set.

Element	Description
Track type	Time signature and style
Data type (* <i>pParam</i>)	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 <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

GUID_Unload

Unloads instrument or wave data. This parameter can be set but not retrieved.

Element	Description
Track type	Band, wave
Data type (* <i>pParam</i>)	IDirectMusicPerformance8 interface
	pointer
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicSegment8::Unload

GUID_UnloadFromAudioPath

Unloads instrument or wave data. This parameter can be set but not retrieved.

Element	Description
Track type	Band, wave
Data type (* <i>pParam</i>)	IDirectMusicAudioPath8 interface
	pointer
mtTime	Not used

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• IDirectMusicSegment8::Unload

GUID_Valid_Start_Time

Retrieves the next valid point within a segment at which it can start. This parameter can be retrieved but not set.

Element	Description	
Track type	Marker, motif, pattern	
Data type (* <i>pParam</i>) DMUS_VALID_START_PARAM		
mtTime	Not used	

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>Segment Timing</u>

GUID_Variations

Retrieves the variations in effect across <u>performance channels</u>. This parameter can be retrieved but not set.

Element	Description	
Track type	Pattern	
Data type (* <i>pParam</i>) DMUS_VARIATIONS_PARAM		
mtTime	Not used.	

Remarks

For an explanation of the table, see <u>Standard Track Parameters</u>.

Requirements

Header: Declared in dmusici.h.

DirectMusic Enumerated Types

This section contains references for the following enumerated types.

Enumerated Type	Description
DMUS_CHORDKEYF_FLAGS	Used in the bFlags member of the DMUS structure.
DMUS_CLOCKTYPE	Used in the ctType member of the DMUS_CLOCKINFO8 structure.
DMUS_COMMANDT_TYPES	Used in the <i>wCommand</i> parameter of the IDirectMusicComposer8::AutoTransitio IDirectMusicComposer8::ComposeTran and in the <i>bCommand</i> member of the DMUS_COMMAND_PARAM structure.
DMUS_COMPOSEF_FLAGS	Used in the <i>dwFlags</i> parameter of the IDirectMusicComposer8::AutoTransitio IDirectMusicComposer8::ComposeTran
DMUS_CURVE_FLAGS	Used in the bFlags member of the DMUS_CURVE_PMSG structure.
DMUS_EMBELLISHT_TYPES	Used in the wEmbellishment member of t <u>DMUS_IO_PATTERN</u> structure.
DMUS_NOTEF_FLAGS	Used in the bFlags member of the DMUS structure.
DMUS_PATTERNT_TYPES	Used in various command structures to con patterns are selected in sequential comman
DMUS_PLAYMODE_FLAGS	Used in various structures to specify play n
DMUS_PMSGF_FLAGS	Used in the dwFlags member of the DMU structure.
DMUS_PMSGT_TYPES	Used in the dwType member of the DMUS structure to identify the type of message.
DMUS_SEGF_FLAGS	Passed to various methods of IDirectMusi to control the timing and other aspects of a segment.
	Used in the <i>wShape</i> parameter of the IDirectMusicComposer8::ComposeSegn

DMUS_SHAPET_TYPES	and IDirectMusicComposer8::ComposeTem
	methods to specify the desired pattern of th
	Used in the
DMUS_STYLET_TYPES	IDirectMusicPatternTrack8::SetPattern
	IDirectMusicStyle8::EnumPattern methe
	type of pattern.
	Used in the dwFlags member of the DMU !
DMUS_TIME_RESOLVE_FLAGS structure and in the <i>dwTimeResolveFlag</i>	
	IDirectMusicPerformance8::GetResolve
	Used in the <i>dwFlags</i> parameter of the
DMUS TRACKF FLAGS	IDirectMusicTrack8::Play and
	IDirectMusicTrack8::PlayEx methods.
DMUS_VARIATIONT_TYPES	Used in the DMUS_IO_PARTREF structu
	way variations are selected in sequential co

DMUS_CHORDKEYF_FLAGS

The **DMUS_CHORDKEYF_FLAGS** enumerated type is used in the **bFlags** member of the **DMUS_CHORD_KEY** structure.

Syntax

```
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.

See Also

• <u>DirectMusic Enumerated Types</u>

DMUS_CLOCKTYPE

The **DMUS_CLOCKTYPE** enumerated type is used in the **ctType** member of the **DMUS_CLOCKINFO8** structure.

Syntax

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 waveform-playback device.

Requirements

Header: Declared in dmusicc.h.

See Also

• <u>DirectMusic Enumerated Types</u>

DMUS_COMMANDT_TYPES

The **DMUS_COMMANDT_TYPES** enumerated type is used in the *wCommand* parameter of the **IDirectMusicComposer8::AutoTransition** and **IDirectMusicComposer8::ComposeTransition** methods and in the **bCommand** member of the **DMUS_COMMAND_PARAM** structure.

Syntax

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_END = 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.

See Also

• <u>DirectMusic Enumerated Types</u>

DMUS_COMPOSEF_FLAGS

The **DMUS_COMPOSEF_FLAGS** enumerated type is used in the *dwFlags* parameter of the **IDirectMusicComposer8::ComposeTransition** and **IDirectMusicComposer8::AutoTransition** methods.

Syntax

typedef enum enumDMUS_COMPOSEF_FLAGS {			
DMUS_COMPOSEF_NONE	=	Θ,	
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	=	0×80,	
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	=	0×8000,	
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_INVALIDATE_PRI	=	0x400000	
<pre>} DMUS_COMPOSEF_FLAGS;</pre>			

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*. Not implemented.

DMUS_COMPOSEF_IMMEDIATE

AutoTransition only. Start transition immediately.

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 <u>style</u> 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.

DMUS_COMPOSEF_INVALIDATE_PRI

Invalidate only the primary segment when transitioning to a new segment.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Enumerated Types</u>
- <u>DMUS_SEGF_FLAGS</u>

DMUS_CURVE_FLAGS

The **DMUS_CURVE_FLAGS** enumerated type is used in the **bFlags** member of the **DMUS_CURVE_PMSG** structure.

Syntax

```
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.

See Also

• <u>DirectMusic Enumerated Types</u>

DMUS_EMBELLISHT_TYPES

The **DMUS_EMBELLISHMENT_TYPES** enumerated type is used in the **wEmbellishment** member of the **DMUS_IO_PATTERN** structure.

Syntax

```
typedef enum enumDMUS_EMBELLISHT_TYPES {
    DMUS_EMBELLISHT_NORMAL =
                                 Θ,
                                1,
   DMUS_EMBELLISHT_FILL =
   DMUS_EMBELLISHT_BREAK =
                                 2,
   DMUS EMBELLISHT INTRO =
                                 4,
   DMUS_EMBELLISHT_END =
                                8,
   DMUS_EMBELLISHT_MOTIF =
                                16,
   DMUS EMBELLISHT ALL
                          = 0xFFFF
} DMUS_EMBELLISHT_TYPES;
```

Constants

DMUS_EMBELLISHT_NORMAL

Normal pattern.

DMUS_EMBELLISHT_FILL

Fill pattern.

DMUS_EMBELLISHT_BREAK

Break pattern.

DMUS_EMBELLISHT_INTRO

Intro pattern.

DMUS_EMBELLISHT_END

End pattern.

DMUS_EMBELLISHT_MOTIF

Motif pattern.

DMUS_EMBELLISHT_ALL

Combination of all types.

Requirements

Header: Declared in dmusicf.h.

See Also

• <u>DirectMusic Enumerated Types</u>

DMUS_NOTEF_FLAGS

The **DMUS_NOTEF_FLAGS** enumerated type is used in the **bFlags** member of the **DMUS_NOTE_PMSG** structure.

Syntax

```
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 <u>music value</u>, 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.

See Also

• DirectMusic Enumerated Types

DMUS_PATTERNT_TYPES

The **DMUS PATTERNT TYPES** enumerated type is used in various command structures to control the way patterns are selected in sequential commands.

Syntax

```
typedef enum enumDMUS_PATTERNT_TYPES {
                       = 0,
= 1,
  DMUS_PATTERNT_RANDOM
 DMUS_PATTERNT_REPEAT
 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

- <u>DirectMusic Enumerated Types</u>
- DMUS_COMMAND_PARAM_2
- <u>DMUS_COMMAND_PARAM</u>
- DMUS_IO_COMMAND
- <u>DMUS_VARIATIONT_TYPES</u>

DMUS_PLAYMODE_FLAGS

The **DMUS PLAYMODE FLAGS** enumerated type is used in various structures to specify play modes. The play mode determines how a music value is transposed to a MIDI note.

Syntax

```
typedef enum enumDMUS_PLAYMODE_FLAGS {
 DMUS_PLAYMODE_KEY_ROOT
                            = 1,
= 2,
 DMUS_PLAYMODE_CHORD_ROOT
 DMUS_PLAYMODE_SCALE_INTERVALS = 4,
 DMUS PLAYMODE CHORD INTERVALS = 8,
                                = 16,
 DMUS PLAYMODE NONE
```

```
} 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

- <u>DirectMusic Enumerated Types</u>
- <u>DMUS_IO_STYLENOTE</u>
- <u>DMUS_IO_STYLEPART</u>
- <u>DMUS_NOTE_PMSG</u>
- IDirectMusicPerformance8::MIDIToMusic
- IDirectMusicPerformance8::MusicToMIDI
- Music Values and MIDI Notes

DMUS_PMSGF_FLAGS

The **DMUS_PMSGF_FLAGS** enumerated type is used in the **dwFlags** member of the **DMUS_PMSG** structure.

Syntax

```
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

See Also

- <u>DirectMusic Enumerated Types</u>
- IDirectMusicPerformance8::SendPMsg
- <u>IDirectMusicTool8::GetMsgDeliveryType</u>

DMUS_PMSGT_TYPES

The **DMUS_PMSGT_TYPES** enumerated type is used in the **dwType** member of the **DMUS_PMSG** structure to identify the type of message.

Syntax

```
typedef enum enumDMUS_PMSGT_TYPES {
  DMUS_PMSGT_MIDI = 0,
                        = 1,
  DMUS PMSGT NOTE
  DMUS_PMSGT_SYSEX = 2,
                             = 3,
  DMUS PMSGT NOTIFICATION
  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 **<u>DMUS_CURVE_PMSG</u>**.

DMUS_PMSGT_TIMESIG

Time signature. See **<u>DMUS_TIMESIG_PMSG</u>**.

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 **<u>DMUS_CHANNEL_PRIORITY_PMSG</u>**.

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 waveform. See **DMUS_WAVE_PMSG**.

DMUS_PMSGT_LYRIC

Lyric message. See **DMUS_LYRIC_PMSG**.

DMUS_PMSGT_SCRIPTLYRIC

Lyric message sent by a script. See **<u>DMUS_LYRIC_PMSG</u>**.

DMUS_PMSGT_USER

User-defined message.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>DirectMusic Enumerated Types</u>

DMUS_SEGF_FLAGS

The **DMUS_SEGF_FLAGS** enumerated type is passed to various methods of **IDirectMusicPerformance8** to control the timing and other aspects of actions on a segment.

Syntax

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_INVALIDATE_PRI	= 1<<28		
<pre>} DMUS_SEGF_FLAGS;</pre>			

Constants

DMUS_SEGF_REFTIME

Time parameter is in reference time.

DMUS_SEGF_SECONDARY

Secondary segment.

DMUS_SEGF_QUEUE

For a primary segment, play at the end of the primary segment queue. For a secondary segment, play at the end of the secondary segment specified in the *pFrom* parameter of **IDirectMusicPerformance8::PlaySegmentEx**.

DMUS_SEGF_CONTROL

Play as a control segment. Valid for secondary segments only. See Remarks.

DMUS_SEGF_AFTERPREPARETIME

Resolve time to a time after the prepare time. See **IDirectMusicPerformance8::GetPrepareTime**.

DMUS_SEGF_GRID

Resolve time to a grid boundary.

DMUS_SEGF_BEAT

Resolve time to a beat boundary.

DMUS_SEGF_MEASURE

Resolve time to a measure boundary.

DMUS_SEGF_DEFAULT

Use flags embedded in the segment. This resolves the time to the segment's default boundary and also causes the segment to play on its embedded audiopath, if it was configured to do so in the authoring application.

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

Resolve time to a time after the queue time. This is the default for primary segments. Ignored if DMUS_SEGF_AFTERPREPARETIME is also set.

DMUS_SEGF_AFTERLATENCYTIME

Resolve time to a time 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. If the new segment is being played as a primary segment, any primary segments already queued after the currently playing primary segment are flushed. If no primary segment is playing, use other resolution flags. When combined with DMUS_SEGF_ALIGN, this flag causes the beginning of the cued segment to be aligned with the beginning of the current primary segment.

DMUS_SEGF_MARKER

Resolve time to the 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.

DMUS_SEGF_INVALIDATE_PRI

Invalidate only the primary segment when transitioning to a new segment.

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 <u>Control Segments</u>.

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 <u>Segment Timing</u>.

Requirements

Header: Declared in dmusici.h.

See Also

- <u>DirectMusic Enumerated Types</u>
- <u>DMUS_TIME_RESOLVE_FLAGS</u>

- IDirectMusicPerformance8::Invalidate
- <u>IDirectMusicPerformance8::PlaySegment</u>
- <u>IDirectMusicPerformance8::PlaySegmentEx</u>
- <u>IDirectMusicPerformance8::Stop</u>
- <u>IDirectMusicSegment8::GetDefaultResolution</u>
- IDirectMusicSegment8::SetDefaultResolution
- <u>Segment Timing</u>

DMUS_SHAPET_TYPES

The **DMUS_SHAPET_TYPES** enumerated type is used in the *wShape* parameter of the **IDirectMusicComposer8::ComposeSegmentFromShape** and **IDirectMusicComposer8::ComposeTemplateFromShape** methods to specify the desired pattern of the <u>groove level</u>.

Syntax

```
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.

See Also

• <u>DirectMusic Enumerated Types</u>

DMUS_STYLET_TYPES

The **DMUS_STYLET_TYPES** enumerated type is used in the **IDirectMusicPatternTrack8::SetPatternByName** and **IDirectMusicStyle8::EnumPattern** methods to specify a type of pattern.

Syntax

```
typedef enum enumDMUS_STYLET_TYPES {
  DMUS_STYLET_PATTERN = 0,
  DMUS_STYLET_MOTIF = 1,
} DMUS_STYLET_TYPES;
```

Constants

DMUS_STYLET_PATTERN

Normal pattern.

DMUS_STYLET_MOTIF

Motif pattern.

Requirements

Header: Declared in dmusici.h.

See Also

• <u>DirectMusic Enumerated Types</u>

DMUS_TIME_RESOLVE_FLAGS

The **DMUS_TIME_RESOLVE_FLAGS** enumerated type is used in the **dwFlags** member of the **DMUS_PMSG** structure and in the *dwTimeResolveFlags* parameter of the **IDirectMusicPerformance8::GetResolvedTime** method.

Syntax

typedef enum enumDMUS_TIME_RESOLVE_FLAGS {
 DMUS_TIME_RESOLVE_AFTERPREPARETIME = DMUS_SEGF_AFTERPREPARETIME,
 DMUS_TIME_RESOLVE_AFTERQUEUETIME = DMUS_SEGF_AFTERQUEUETIME,
 DMUS_TIME_RESOLVE_GRID = DMUS_SEGF_AFTERLATENCYTIME,
 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

- <u>DirectMusic Enumerated Types</u>
- <u>Timing</u>

DMUS_TRACKF_FLAGS

The **DMUS_TRACKF_FLAGS** enumerated type is used in the *dwFlags* parameter of the **IDirectMusicTrack8::Play** and **IDirectMusicTrack8::PlayEx** methods.

Syntax

```
typedef enum enumDMUS_TRACKF_FLAGS {
   DMUS_TRACKF_SEEK = 1,
   DMUS_TRACKF_LOOP = 2,
   DMUS_TRACKF_START = 4,
   DMUS_TRACKF_FLUSH = 8,
   DMUS_TRACKF_DIRTY = 0×10,
   DMUS_TRACKF_NOTIFY_OFF = 0×20,
   DMUS_TRACKF_PLAYOFF = 0×40,
   DMUS_TRACKF_LOOPEND = 0×80,
   DMUS_TRACKF_STOP = 0×100,
   DMUS_TRACKF_RECOMPOSE = 0×200,
   DMUS_TRACKF_CLOCK = 0×400,
} 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.

See Also

• <u>DirectMusic Enumerated Types</u>

DMUS_VARIATIONT_TYPES

The **DMUS_VARIATIONT_TYPES** enumerated type is used in the **DMUS_IO_PARTREF** structure to specify the way variations are selected in sequential commands.

Syntax

```
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:
```

```
} 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

- <u>DirectMusic Enumerated Types</u>
- DMUS_COMMAND_PARAM_2
- <u>DMUS_COMMAND_PARAM</u>
- <u>DMUS_IO_COMMAND</u>
- <u>DMUS_PATTERNT_TYPES</u>

DirectMusic GUIDs

This section lists globally unique identifiers used in DirectMusic. The following topics are covered:

- DirectMusic Interface GUIDs
- <u>DirectMusic Component GUIDs</u>

DirectMusic Interface GUIDs

The following GUIDs are used as interface identifiers (IIDs) to obtain DirectMusic interfaces.

IID	Interface
IID_IDirectMusic8	IDirectMusic8
IID_IDirectMusicAudioPath8	IDirectMusicAudioPath8
IID_IDirectMusicBand8	IDirectMusicBand8
IID_IDirectMusicBuffer8	IDirectMusicBuffer8
IID_IDirectMusicChordMap8	IDirectMusicChordMap8
IID_IDirectMusicCollection8	IDirectMusicCollection8
IID_IDirectMusicComposer8	IDirectMusicComposer8
IID_IDirectMusicContainer8	IDirectMusicContainer8
IID_IDirectMusicDownload8	IDirectMusicDownload8
IID_IDirectMusicDownloadedInstrument8	BIDirectMusicDownloadedInstrumer
IID_IDirectMusicGetLoader8	IDirectMusicGetLoader8
IID_IDirectMusicGraph8	IDirectMusicGraph8
IID_IDirectMusicInstrument8	IDirectMusicInstrument8
IID_IDirectMusicLoader8	IDirectMusicLoader8
IID_IDirectMusicObject8	IDirectMusicObject8
IID_IDirectMusicPatternTrack8	IDirectMusicPatternTrack8
IID_IDirectMusicPerformance8	IDirectMusicPerformance8
IID_IDirectMusicPort8	IDirectMusicPort8
IID_IDirectMusicPortDownload8	IDirectMusicPortDownload8
IID_IDirectMusicScript8	IDirectMusicScript8
IID_IDirectMusicSegment8	IDirectMusicSegment8
IID_IDirectMusicSegmentState8	IDirectMusicSegmentState8
IID_IDirectMusicStyle8	IDirectMusicStyle8
IID_IDirectMusicThru8	IDirectMusicThru8
IID_IDirectMusicTool8	IDirectMusicTool8
IID_IDirectMusicTrack8	IDirectMusicTrack8
IID_IKsControl	IKsControl

IID_IReferenceClock

IReferenceClock

Note Where different versions of an interface are available, the identifier is shown only for the most recent one. Where an interface has not changed from previous versions of DirectX, the IID in the table is defined as the original IID. For example, IID_IDirectMusicInstrument8 is defined as IID_IDirectMusicInstrument in Dmusicc.h, because **IDirectMusicInstrument8** is the same as **IDirectMusicInstrument**.

A few alternative identifiers, such as IID_IDirectMusicPerformance2, are defined for obtaining special implementations of interfaces. Most applications should not use these implementations.

DirectMusic Component GUIDs

The following GUIDS are used as class identifiers (CLSIDs) of DirectMusic components.

CLSID	Component	
CLSID_DirectMusic	DirectMusic object	
CLSID_DirectMusicAudioPathConfig	Audiopath configuration	
CLSID_DirectMusicBand	Band	
CLSID_DirectMusicBandTrack	Band track	
CLSID_DirectMusicChordMap	Chordmap	
CLSID_DirectMusicChordMapTrack	Chordmap track	
CLSID_DirectMusicChordTrack	Chord track	
CLSID_DirectMusicCollection	Collection	
CLSID_DirectMusicCommandTrack	Command (groove) track	
CLSID_DirectMusicComposer	Composer	
CLSID_DirectMusicContainer	Container	
CLSID_DirectMusicGraph	Toolgraph	
CLSID_DirectMusicLoader	Loader	
CLSID_DirectMusicLyricsTrack	Lyric track	
CLSID_DirectMusicMarkerTrack	Marker track	
CLSID_DirectMusicMotifTrack	Motif track	
CLSID_DirectMusicMuteTrack	Mute track	
CLSID_DirectMusicParamControlTrack	Parameter control track	
CLSID_DirectMusicPatternTrack	Pattern track	
CLSID_DirectMusicPerformance	Performance	
CLSID_DirectMusicScript	Script	
CLSID_DirectMusicScriptTrack	Script track	
CLSID_DirectMusicSegment	Segment	
CLSID_DirectMusicSegmentState	Segment state	
CLSID_DirectMusicSegmentTriggerTrackSegment trigger track		
CLSID_DirectMusicSeqTrack	Sequence track	
CLSID_DirectMusicSignPostTrack	Signpost track	

Style
Style track
Synthesizer
Sysex track
Tempo track
Time signature track
Wave track
Wave

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
0x0000000	<u>S_OK</u>
0x0000001	<u>S_FALSE</u>
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	<u>E_FAIL</u>
0x80070057	<u>E_INVALIDARG</u>
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
0x88781129	DMUS_E_NOARTICULATION
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
0x88781134 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	<u>DMUS_E_INVALIDBUFFER</u>
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
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	<u>DMUS_E_SCRIPT_ERROR_IN_SCRIPT</u>
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 <u>DirectMusic Return Values by Number</u>.

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 <u>DLS</u> 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 <u>performance channels</u> cannot be allocated.

DMUS_E_AUDIOPATHS_NOT_VALID

<u>Performance channels</u> have been set up by using <u>IDirectMusicPerformance8::AssignPChannel</u>, 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 <u>DLS</u> collection.

DMUS_E_BADINSTRUMENT

Invalid instrument chunk in <u>DLS</u> collection.

DMUS_E_BADOFFSETTABLE

The offset table has errors.

DMUS_E_BADWAVE

Corrupt wave header.

DMUS_E_BADWAVELINK

The wave-link chunk in <u>DLS</u> 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 <u>DLS</u> 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 <u>DLS</u> 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 <u>DLS</u> 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 <u>DLS</u> collection.

DMUS_E_NOTMONO

The wave chunk has more than one interleaved channel. <u>DLS</u> format requires mono.

DMUS_E_NOTPCM

Waveform 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 <u>DLS</u> 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.